INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Weakest Relative Precondition Semantics

Balancing Approved Theory and Realistic Translation Verification

Andreas Wolf

Bericht Nr. 2013
Februar 2001

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Weakest Relative Precondition Semantics

Balancing Approved Theory and Realistic Translation Verification

Andreas Wolf

Bericht Nr. 2013
Februar 2001

e-mail: awo@informatik.uni-kiel.de

Dieser Bericht enthält die Dissertation des Verfassers

Tag der mündlichen Prüfung: 13. Februar 2001

Referent: Prof. Dr. Dr.h.c. Hans Langmaack
Koreferent: Prof. Dr. Bernhard Steffen

Table of Contents

List of Figures

List of Tables

1. Introduction

The present thesis is another contribution to the well studied theory of program verification and specifically translation verification. Though this particular area is apparently classic there is still a great need for further theoretical considerations like the ones presented here because it turns out that – roughly speaking – the approved theory does not cope with the practical needs. On this account it is advisable to outline the current state of affairs at first.

The overall motivation is the fact that compilers are one of the essential computing environments nowadays. Their use ranges from the traditional translation of higher programming languages over conversions between data formats of a large variety to more recent applications like the generation of WWW contents from data stored in data bases (e.g. in cross media applications like electronic ware houses, web catalogs, search engines etc.). The rather inconspicuous use of compilers helps to get rid of architecture or system specific representations and allows to handle data or algorithms in a more convenient abstract form. Without exaggeration, compilers thus constitute an indispensable core technology for modern information society.

In all these sketched scenarios, however, there is a great demand for correct programs, for correct translations and for trustworthy software in general, be it in the field of safety critical systems or to warrant privacy in our contemporary networked world.

The theory underlying the syntactic aspects of compiler construction is well-understood and documented in quite a number of text books (e.g. [1, 50, 76, 82, 84]). It is easily applied in practice via automated tools like scanner and parser generators. This has made the construction of the syntactic phases of compilers (like scanners and parsers), which has been a challenge back in the sixties and seventies, 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 classic task, the translation of higher programming languages which will play the role of a running example throughout this thesis, there is still no generally followed approach, although some well-studied frameworks like, e.g., action semantics [58] 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 (see, e.g., the above

mentioned textbooks and the more recent [59]). 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, however, 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 assembly-code [43, 72]. Trusted and fully-understood compilers would enable a certification on the source language level which would be less time-consuming, cheaper, and more reliable. From a practical point of view, the ultimate goal of compiler verification [11, 28, 29, 35, 61, 66, 67] should be to improve on this matter.

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. (In our opinion, proofs based on operational semantics are particularly sensitive to this danger because such definitions are by their very nature rather detailed and clumsy.) This calls for the employment of an abstract style of semantics in a compiler proof. (Thus, in a compiler proof an abstract kind of semantics should be used.) On the other hand, it is important that the used semantic description is rather close to the intuition of the average programmer in order to avoid errors resulting from misunderstandings 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 programs, the ultimate reference point should thus be a rather concrete semantics.

How can we resolve the obvious conflict between the requirements of using an operational as well as an abstract kind of semantics? A remedy is 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 alone without directly recurring to the operational definition. A particular benefit of this approach is that the abstract semantics to be used in the compiler proof can be suited to the specific correctness property to be established.

The most popular representatives in this regard are Dijkstra's so-called wp- and wlp-transformers [17, 18] which provide abstractions of an underlying operational or relational semantics and which allow a reasoning about program behavior in a denotational, i.e. compositional, manner in the framework of predicates and

monotonic predicate transformers, i.e. on a state-free level.[1] Colloquially speaking, for a program π, the *weakest precondition* predicate transformer $\mathsf{wp}.\pi$ suits to a so-called notion of *total correctness* of π and the *weakest liberal precondition* predicate transformer $\mathsf{wlp}.\pi$ to a so-called notion of *partial correctness* of π. What distinguishes $\mathsf{wp}.\pi$ from $\mathsf{wlp}.\pi$ and consequently total from partial correctness is *regular (successful) termination*. For the purpose of this motivation, however, it suffices to note that a program is totally correct w.r.t. a precondition ϕ and a postcondition ψ iff it is partially correct w.r.t. these conditions – i.e. if π starts in a state satisfying ϕ and if π delivers a regular result at all then this result satisfies ψ – and if it terminates regularly for all initial states satisfying the precondition ϕ.[2] Consequently, a state satisfies the weakest precondition of π w.r.t. postcondition ψ, i.e. $\mathsf{wp}.\pi.\psi$, iff it satisfies the weakest liberal precondition of π w.r.t. postcondition ψ, i.e. $\mathsf{wlp}.\pi.\psi$, and if π terminates regularly if started in this state, i.e. if $\mathsf{wp}.\pi.\mathtt{true}$ holds. The advantage of a predicate transformer reasoning over a reasoning in terms of an operational semantics is that one can make use of the fact that the underlying spaces of predicates and predicate transformers are lattices, frameworks which are equipped with an order, namely the implication order resp. a lifted version thereof, and other connectives. Furthermore and in contrast to an operational semantics, each predicate transformer directly corresponds to a specific correctness property and typically those predicate transformers are defined in an inductive manner which allows to infer correctness properties of a program from its components.

But proving programs correct – in which sense ever – is not worth while if the compiler which actually generates the executable is erroneous itself. Based on the mentioned program correctness notions it is proximate to say that a compiler, or a translation, preserves total resp. partial correctness if all total resp. partial correctness assertions made for the source program transfer to the target program. This is a nice, elegant and approved way of defining translation correctness but, unfortunately, it turns out that no "realistic compiler" running on a "real machine" can ever preserve total or partial correctness!

The reasons for this dilemma are twofold. Firstly, resource limitations and the restricted arithmetic of the execution mechanism may, e.g., lead to spontaneous aborts. It is obvious – and everyone involved in programming will have encountered these situations – that a program running on a real machine aborts with an error message like, e.g., "StackOverflow", "OutOfMemory" or "ArithmeticError". Thus, even if a program has been proved totally correct on the source level the generated target program running in the real world may not be totally correct as it may fail; consequently a compiler cannot preserve total correctness. Secondly, and rather surprisingly, a translation cannot preserve partial correctness either because common and ubiquitous code optimizations may let an optimized programs behave chaotically and hence violate the specification of the source program. Consider, for instance, the program

[1] In [17, 18] those transformers are given in an axiomatic fashion but for the moment they can nevertheless be kept for derived terms.

[2] We refer to the pair of pre- and postcondition which assure π's correctness also as π's *specification*.

$$\pi \;=\; x := e \;;\, x := f \;;\, P \;\;,$$

where P is an arbitrary program. It is intuitively safe to remove the first assignment from π if f does not depend on the value of x because x is immediately overwritten by the second assignment. The so-called *dead code elimination* (or redundant code elimination) strategy proceeds this way and replaces π by

$$\pi' \;=\; x := f \;;\, P \;\;.$$

Now assume that expression e is a division by zero, say $e = 1/0$. Then the initial program π is partially correct for trivial reasons – it does not terminate regularly but aborts propagating a, say, "DivByZero"-error – whereas the optimized program may behave arbitrarily depending on the shape of P. It may particularly violate π's specification because π is partially correct w.r.t. every specification such that each specification violated by π' is a simple example.

Summarily, there is a gap between reality and theory in the following sense. On the one hand, in order to cope with businesslike translations of authentic programs to executables running on existing machines, an abstract kind of semantics is quite desirable because this would hopefully ease the verification exercise and increase both reliability and trust in the proof. But on the other hand the nice and approved theory of program and particularly translation verification – which indeed provides an abstract and handy framework – cannot manage reality as it fits to straightline translations of toy-languages running on idealized machines only.

However, neither the demands of reality nor the weakest precondition semantics in general should be blamed for this sad state of affairs but solely the classic notions in the field of program and translation verification, namely partial and total correctness resp. preservation thereof, and the indiscriminate idewntification of aborts and divergence.

Some deeper and unprejudiced reflections on translations unveil that there is a variety of other preservation properties that can be of interest. A compiler, for instance, which preserves termination behavior in the sense that the generated target programs do not spontaneously diverge might well be called "correct" even if the target program aborts propagating an "OutOfMemory"-error because this is not the fault of the compiler but of the machine. As another example consider an optimizing compiler which is allowed to generate diverging target programs; this is of particular interest because a termination proof for the source program is dispensable. Preservation of partial correctness cannot be achieved but if the source program is shown to produce no arithmetic errors then a dead-code-elimination as sketched in the above example is permissible and the translation is "correct" in some sense. Furthermore, it might be of particular interest that not only regular results are preserved but also finite errors, e.g. for debugging purposes a "DivByZero"-error observed on the target level should indeed be produced by a division by zero on the source level. Note that none of these sketched scenarios can be adequately handled in the classical setting because both total and partial correctness resp. preservation of total and preservation of partial correctness identify aborts with divergence and focus on preservation of some regular states.

It becomes clear that these questions of practical concerns can only be managed with a more careful distinction between different erroneous results and divergence together with the causes for their occurrences. The proposal presented here which is intended to serve for a remedy is the following: The set of all possible, say, outcomes, i.e. the set of regular states together with the set of finite errors and divergence, is partitioned into three sets PO, AO and CO, the members of which have to be preserved literally (typically this is the set of regular results but even finite errors may be contained as motivated before), which are accepted (characteristically those errors which are due to the violation of some resource limitations) resp. which are rejected (all others which, e.g., may let the target program behave chaotically). Then a target program can be called a *correct implementation* of a source program w.r.t. a given partitioning if for every initial state each result obtained by the target program is also possible for the source program, i.e. it is contained in PO, or it is accepted, i.e. a member of AO, or there exists a source program computation starting in this initial state which behaves chaotically in the sense that it does not satisfy its specification and consequently the target program need not to implement the source program on this input. Thus, each partitioning gives rise to a very precise and specific translation correctness property which serves and supports the demands of practice. In fact, each of the correctness properties sketched above can be expressed by means of a particular partitioning.

Obviously, the well elaborated theory of $\mathsf{wp}.\pi$ and $\mathsf{wlp}.\pi$ cannot adequately assist reasoning about these family of correctness notions but it is a surprisingly unspectacular step to widen their definitions in such ways that they indeed can. Instead of *rejecting all erroneous outcomes*, like $\mathsf{wp}.\pi$, resp. *accepting all erroneous outcomes*, like $\mathsf{wlp}.\pi$, only some of them, ones contained in a set, say, A, are accepted and the others are rejected. This more discriminate differentiation allows a specification of a so-called *weakest relative precondition predicate transformer* $\mathsf{wrp}_A.\pi$ which harmonizes with the approved theory well in the sense that $\mathsf{wp}.\pi$ and $\mathsf{wlp}.\pi$ are just the border cases of $\mathsf{wrp}_A.\pi$. No new theory has to be developed, only the interpretation changes and, thus, as much as possible from the elegant appeal of the traditional idealized setting is preserved while the more practical questions mentioned before can be considered, too. In particular, the notion of a correct implementation can nicely be expressed in terms of wrp_A-transformers.

To hit the spot, the proposed wrp_A-transformers are intended to bridge the gap between approved theory, i.e. the theory of monotonic predicate transformers in connection with the theory of weakest precondition semantics, and the practical demands of realistic translation verification, namely a correctness notion which copes with "businesslike", i.e. optimizing, compilers generating executables running on real machines. They provide an abstract kind of semantics which, on the one hand, promises to facilitate proper translation verification exercises on a comprehensible and credible level but which, on the other hand, have an operational and thus transparent background such that doubtfulness and errors due to misunderstanding hopefully become rare. The actual contribution is the introduction

to the field of weakest relative precondition semantics and the justification that it indeed constitutes an enrichment.

1.1 Organization of this Thesis

We invite the reader to partake in the following tour. The skin-deep insight that the classical theory of program and translation verification does not apply in reality is really worth some further considerations. Therefore, Chap. 2 is devoted to a more profound discussion on this matter and we stepwisely derive an adequate notion of a *correct implementation* that resist the counterexamples which demonstrate that others are not appropriate in reality; it thereby serves the second part of the subtitle. Chap. 3 keeps up with the very basics of lattice theory and related topics. Based on further reflections on the classical setup, namely partial and total correctness resp. preservation thereof, Chap. 4 introduces the notion of *relative correctness* w.r.t. a set A of outcomes to be accepted and the corresponding family of wrp_A-transformers. Since those transformers can, if properly put together, express fine-grained implementation correctness properties on an abstract level, they indeed affirm the first part of the subtitle.

In Chap. 5 the situation becomes yet a bit more real. As a translation typically includes a change of data, e.g. source code text vs. executable binary, it is shown how wrp_A-based reasoning gets along with programs running on different state spaces. Moreover, to support the practitioner's needs and for realistic translation verification purposes it is important to study how correctness preservation properties of translations transfer if they are composed vertically or sequentially.

In [17, 18] the wp- and wlp-transformers are defined axiomatically and in some sense Chap. 6 follows this approach. Apart from some basic algebraic properties of the derived wrp_A-transformers it is shown that it is possible to switch between an *axiomatically defined* predicate transformer semantics and a relational semantics without loss of information, i.e. both semantics representations are of equal expressiveness.

To embrace the title of the present thesis, Chap. 7 is concerned with weakest relative precondition semantics of programming languages. We consider an abstract assembly language, the programs of which are assumed to be executed on a finite machine, and a simple WHILE-language equipped with parameterless procedures. They are intended to cover most of the customary languages and allow to study the essences of the control flow aspects of assembly resp. ALGOL-like languages. The actual benefit of this chapter is the derivation of a wrp_A-semantics from an operational one; just like proposed and motivated before.

Finally, it is shown in Chap. 8 that wrp_A-based reasoning indeed keeps the promise to balance the gap between theory and practice of translation verification. By example some common code optimizations are visited and shown to be correct under certain and well defined requirements. These considerations underpin the claim that a weakest relative precondition semantics is able to cope with realistic translations because Chap. 2 demonstrates that code optimizations may

forbid preservation of partial correctness in the real world but the idealized setting allows to prove them correct anyhow. To complete the running example a translation of high-level programs to assembly code is proved correct, and this in two respects. In a first case the termination behavior has to be preserved unless the executing machine cannot stack all needed return addresses. It shows that an interesting case like this can be managed in the less restricted world of wrp_A-transformers; remember that a translation cannot preserve total correctness but it might well be desirable to preserve the termination behavior. In a second example spontaneous divergence is allowed, too, because this is of particular interest for compiler verification purposes. In fact, this second case is the more mentionable one because we know of no clean proof which considers procedures and preservation of partial correctness, not to mention variations thereof.

The Appendix is attached for, say, historic reasons. It claims that the abstract assembly language presented in Sect. 7.3 is *almost* an abstract view on an existing language, namely the Transputer-code [37], and together with the primary [61] it justifies our understanding of the assembly language being nearly taken from reality and not purely artificial.

1.2 Related Work

This monograph ranges in a variety of akin topics. It has its roots in the *Verifix* project [23], a ProCoS follow-up [12], which is supported by the Deutsche Forschungsgemeinschaft (DFG) since 1995. The three research groups from the universities of Karlsruhe (site leader: G. Goos), Kiel (site leader: H. Langmaack) and Ulm (site leader: F. v. Henke) have made it their business to work on verified and correctly implemented compilers for realistic programming languages running on existing machines. The ultimate goal is to construct efficient compilers using common techniques in such ways that the generated programs are trustworthy. Essential in this regard is the observation that preservation of partial correctness resp. a variation thereof is the key to keep things manageable. The reader is referred to [20] for a more profound discussion of the current state of affairs.

A vast number of contributions to the field of weakest precondition semantics, e.g. [17, 18, 31], and the refinement calculus, e.g. [4, 55, 56], has been published in the last decades but only a few of those pay attention to procedures, e.g. [7, 8, 32]. What they all have in common and what particularly distinguishes our work from theirs is that we do not restrict to total or partial correctness but consider an entire family of weakest relative precondition semantics. We know of two reports [39, 54] which claim to unify wp and wlp but this only for notational and calculational purposes; as common, aborts are identified with divergence.

Of course, much research has been performed in the area of translation verification (let us mention [51] as the origin). In some sense, we follow the lines of [34, 35, 73] and specifically [60] which also reason about compiler correctness in an algebraic fashion. There is also [47] which considers procedures, too, but all those contributions deal with preservation of total correctness in an idealized

world without finite errors. Code optimizations on the other hand are typically not verified but only, say, verbally justified (see the classical text books and [59]). An exception is the recent [41] which proves optimizations correct in an algebraic style but it is noteworthy that those proofs do not transfer to reality if finite errors are present (see the comments on p. 166).

Finally, we like to stress that the present thesis is heavily inspired by [61] which proves – totally – correct the translation of a timed WHILE-language to an existing processor, namely the Transputer, in an abstract and modular fashion. In some sense our understanding of the behavior of the assembly language, parts of the total-correctness proof and the request to consider preservation of partial correctness stems from this very worth reading monograph.

Acknowledgments

The *Verifix* project has made my dissertation possible: The fruitful ambiance of three research groups, each of which thinking about translation and verification in its very own way, provided a rich and heterogeneous pool of experiences and opinions to profit from. I thank all my *Verifix* colleagues for productive criticism and discussions and I am particularly grateful to the DFG for the financial support.

A lot of people have made my daily life in the office pleasant, comfortable and sometimes even sufferable. Credits in this regard go to the former 2nd floor of the "Dreiecksplatz", the current 7th floor of the "Hochhaus" and in particular to Barbara, Claudia, Frank, Thorsten, Ulf, Ulrich, Ulrike and Willi. During my time at these locations several persons stood by my side and listened to my numberless questions; I would like to thank Rudolf Berghammer, Wolfgang Goerigk and Burghard von Karger for lending me their ears and for their firm willingness to help.

My sincerest thanks are devoted to those without whom this thesis would not have been realizable at all. The congenial and gainful collaboration with Markus Müller-Olm laid the foundation stone and was the impulse for – often even the key to – many a consideration. I am thankful for the many ideas and improvements and specifically for having the possibility to share in his know-how. Without the constant encouragement and support of my supervisor Hans Langmaack the present dissertation would not have been within reach either. His widespread experience and knowledge provided the environment for effective research and beyond a lot of hints and suggestions I am particularly grateful to him for enduring my own interests and the way I followed them. Last but not least I thank Renate, Rüdiger, Hildegard, Gerhard and especially Emma, Jeppe and Wiebke for their steady backing.

2. On Correct Translations – A Survey

This chapter is devoted to a more profound motivation and to an illustrating journey through the field of translation correctness. Though this area is apparently well known one should be quite aware of the meaning of the word *correctness* and one should also mind what semantic relationship one *reasonably* can expect to hold between a target program running on a real machine and a source program from which it was generated.

Starting from a very naive image of a correct translation stepwise increasingly more realistic views are discussed and, finally, the journey ends in a hopefully comprehensible definition of a so-called *correct implementation*, a notion which is intended to be fairly adequate for realistic translations running on real machines but which is neither buried behind a mountain of technicalities nor a mere collection of flowery phrases. Moreover, this notion will guide us through the remainder of this thesis in the sense that oncoming discussions concerning so-called *relative correctness* and a family of corresponding *transformers* which facilitate reasoning about it – in fact, this is the actual story told here – are inspired by the truly and surprisingly skin deep insights presented in this chapter.

Most of the following observations were already discussed in [62]. It has a somewhat "tutorial" flavor and the following exposition has it even more as meanwhile some more ideas popped up and this is the place to go further into the question.

2.1 Preliminaries and Nomenclature

Let us first of all set the stage for the oncoming discussion and in particular for the greater part of this monograph. Assume given a set, Π, of programs, π. The reader should imagine imperative, strictly transformational programs intended to compute on a certain non-empty set, Σ, a state-space, the members of which are called (regular) states. Typically, a state is a mapping from variables to values and keeps the current values of the variables in question. A computation of π starts in a state; it represents the input to the program. Of course the computations is expected to deliver an output in the form of a state – for the moment the details of program execution are of no further interest – but sadly the real world is not that simple. Instead, the program may also terminate irregularly, i.e. abort, with a run-time error and it may even diverge, i.e. run forever. Worse as it could be, there may be even more than one *outcome*[1] for one and the same input as programs may

[1] We use the more neutral word "outcome" instead of "result" because some people object to the idea that divergence is a result of a computation.

be non-deterministic. An adequate means for describing the behavior of this kind of programs is a *relational semantics*. Therefore, we assume that each program π is furnished with a relation

$$R(\pi) \subseteq \Sigma \times (\Sigma \cup \Omega) \ . \tag{2.1}$$

Here, Ω is a set disjoint from Σ containing the run-time errors just mentioned, e.g. "DivByZero", "ArithmeticError" etc., and an additional symbol '∞' representing divergence. The following conventions for the naming of variables are used: The set of regular states, Σ, is ranged over by s, the set of erroneous outcomes, Ω, by ω and the set of all outcomes, $\Sigma \cup \Omega$, by σ. The set $\Omega - \{\infty\}$ of so-called finite run-time errors will also be ranged over by o.

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 the finite error o can be reached from s, i.e. π may abort from s propagating o, and $(s, \infty) \in R(\pi)$ says that π may diverge from s. The relational semantics $R(\pi)$ can be thought to be given directly or to be derived from an underlying operational semantics, no matter. We will practice this way in Sect. 7.1 where we also show how the '∞'-symbol enters the relation which seems to be somewhat artificial at first sight.

As any practical program has at least one computation from any given initial state, we might safely assume that $R(\pi)$ is *total*, i.e.

$$\forall s \in \Sigma :: \ \exists \sigma \in \Sigma \cup \Omega :: \ (s, \sigma) \in R(\pi) \ , \tag{2.2}$$

saying that there is an outcome σ with $(s, \sigma) \in R(\pi)$ for all $s \in \Sigma$. However, we do not insist on totality because, firstly, this property is not needed in the sequel (unless otherwise mentioned) and, secondly, each non-total relation can be made total by adding one more special symbol, e.g. '$\dagger$',[2] to the set Ω of irregular outcomes and letting $(s, \dagger)$ be contained in $R(\pi)$ for every initial state s for which π may deliver no outcome (not even diverges). Note that the symbol '$\dagger$' has thus a quite different impact. It stands for undefinedness in the sense of the word: If $(s, \dagger) \in R(\pi)$ then π may deliver *no outcome* if started in s. Some people prefer to call a program undefined if it delivers *no result*, i.e. it is also undefined if it diverges or aborts. It is important to keep in mind that we do not share this concept for the remainder. However, we will be rather careful to indicate all points in the argumentation where certain properties of the underlying relational semantics are essential. The reader is to accept a phrase – namely total correctness, cf. Def. 4.2.2 – just once where totality would be required in order to match the classical meaning of this notion.

Obviously, we consider non-deterministic programs as the $R(\pi)$s are relations. Nevertheless, certain properties of relations and programs will be discussed and one has to agree upon a common understanding. A program π is said to be *univalent* if its relational semantics is, i.e.

$$\forall s, \sigma, \sigma' : \ (s, \sigma) \in R(\pi) \wedge (s, \sigma') \in R(\pi) : \ \sigma = \sigma' \ , \tag{2.3}$$

[2] The classically chosen symbol '$\perp$' will soon get an independent meaning on its own.

or vocalized, if all possible outcomes – if there are any – of π started in an initial state are equal. Finally, program π is said to be *deterministic* if $R(\pi)$ is a function, i.e. if $R(\pi)$ is both total and univalent.

Before going into details of our discussion we would like to make the following remarks which are intended to relate our understanding of a relational semantics to some others considered in the literature. Hoare and Lauer [36] represented programs by relations in the shape of $R(\pi) \subseteq \Sigma \times \Sigma$ such that the possibility of aborting or non-terminating runs is not recorded in the model. This kind of the semantics is an angelic one and fits to partial correctness. Plotkin [70] suggests to insert pairs $(s, \perp)$ if execution of a program started in s may lead to non-termination and in his model all pairs (s, s') are irrelevant if $(s, \perp)$ exists. This is a demonic view and thus it suits to reason about total correctness only. Wand [83] proposes omitting pairs (s, s') for all s' if computations starting in s may not terminate, and Smyth [77] on the other side proposes to insert pairs (s, s') for all s' if execution starting in s may leave to diverging runs. The latter two approaches make the possibility of non-termination indistinguishable from the guarantee of non-termination. Moreover, the presence of so-called finite errors is generally neglected so programs are assumed either to terminate regularly or to diverge. As we have a most general view on the behavior – we record each possible outcome and do not overwrite or insert certain pairs – we have not foreclosed any specific choice of correctness notion and we are enabled to reason about all kinds of regular and erroneous outcomes. Thus, in some sense we have an erratic semantics in mind.

2.2 The Quest for an Adequate Notion

After these preparations let us come to what this chapter is all about and let us discuss what relationship between a source and a target program we might sensibly expect to hold. 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 (the functional programs can be kept for procedure declarations and are used only for denotational convenience). For simplicity we assume that π and π' operate on the same state space. This is of course an unrealistic assumption but representing source program data by target program data just burdens the notation and is not illuminating the control-flow aspects studied here. This will become clearer in Sect. 5.1 which is concerned with data-representations in connection with translation correctness.

If π' is to be a correct implementation of π, we clearly expect that the computations of π' are sensibly 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. 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 discussion, only strictly transformational programs are considered here. 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 translations.

Our very first intuition might be to require that π' and π yield the same outcomes for any given initial state. In other words we might expect π' and π to do exactly the same, formally

$$R(\pi') = R(\pi) \ ,$$

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'. This is obvious if the language in question is equipped with a nondeterministic choice operator, say '|' , the semantics of which is given by $R(\pi_1 \mid \pi_2) = R(\pi_1) \cup R(\pi_2)$. But even more common languages allow inconspicuous nondeterminism. Assume, for instance, 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 variation of value n for variable x in state s, see (7.6). The generated code, π', on the other hand, might well provide the deterministic result 17, as it allocates for z the memory location previously used for y, which still contains y's old value, i.e. 17. No sensible means can enforce full non-determinism in the target code.

What we have learned from this little example is that an equality of relational semantics is far beyond to what we can expect and that we, thus, should at most hope for a relational inclusion to hold, i.e.

$$R(\pi') \subseteq R(\pi) \ .$$

This is the very idea of *refinement*: Each outcome produced by the target program is a possible outcome of the source program (if started in the same initial state). But, again, it turns out that reality thwarts our plans once more: Firstly by limitations of the execution mechanism and secondly by common code optimizations. Let us discuss each of these reasons in turn and demonstrate by means of small examples why even a relational inclusion may be a too strong requirement.

Limited abilities of the implementation might give rise to failure outcomes of the target program that are not possible for the source program at all. One example is restricted arithmetic. Consider, for instance, the following program computing the factorial of a natural number n.

$$\mathsf{fac}(n) \ = \ \mathsf{if}\ n == 1\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathsf{fac}(n-1)$$

Irrespective of the chosen number-representation of the executing processor (except for some computer-algebra systems) one will always be able to find an input n such that an error like "ArithmeticError" or "ArithmeticOverflow" will occur occasionally. But notice that we should be rather glad to observe this error-message at all. The arithmetic overflow has been noticed anyway and the program terminated propagating the error. For performance reasons this might have been omitted and the restricted arithmetic would allow to produce negative numbers as a regular result without even mentioning the overflow! Indeed, this would really fool the user. Another example is finiteness of the memory. Full implementation of recursion, for instance, requires stacks of unbounded size. Actual – and probably future – computers, however, provide only a finite amount of memory; we must thus be prepared to accept outcomes like "StackOverflow" or "OutOfMemory" every now and then when executing programs from languages with unrestricted use of recursion. The Ackermann-function

$$\begin{aligned}\mathsf{ack}(n,m) \;=\; & \text{if } n == 0 \text{ then } m+1 \\ & \text{else if } m == 0 \text{ then } \mathsf{ack}(n-1,1) \\ & \text{else } \mathsf{ack}(n-1,\mathsf{ack}(n,m-1))\end{aligned}$$

is an example for a recursive function which needs a large amount of stack space resp. which has an enormous recursion depth. One might find instances that fail due to a "StackOverflow" rather than an "ArithmeticOverflow".

Such limitations could be handled in various ways. Firstly, one 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, one could simply enrich the source language semantics by those error outcomes which would allow them as possible results of the implementation. This would amount to considering

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

the semantics of π. Thirdly, one 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 solution proposed later will somehow have the flavor of the second approach.

Let us now have a brief but more precise look at the field of code-optimizations. Strange as it may seem, but optimizations can replace error outcomes by arbitrary outcomes. As a first example consider the innocuously looking transformation pictured in Fig. 2.1, an instance of what is called *dead code elimination* [59].[3] 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

[3] The name of this optimization strategy might be misleading: The code to be eliminated is not dead in the sense that it is not reachable by any possible computation but in the sense that it is redundant, i.e. not worth to be executed. However, this is the common parlance.

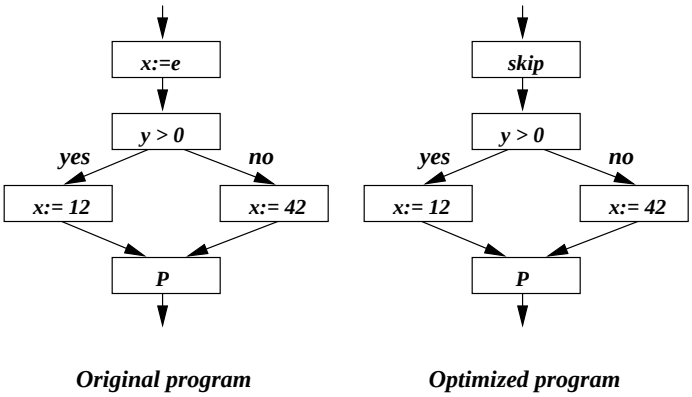

Fig. 2.1. Elimination of dead code.

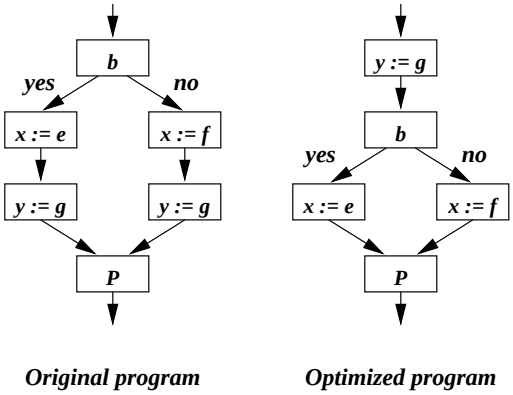

Fig. 2.2. A code motion transformation.

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 an error outcome, say "DivByZero", while the right program can have, depending on P, whatever outcomes you want![4]

As a second example of an optimization consider the *code motion transformation* [59] in Fig. 2.2 where b, e and f are assumed not to contain y, and g is assumed not to contain x, i.e. evaluation of the expressions does not depend on the values of the corresponding variables. 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 reason 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) and g does not depend on the prior assignment. 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 "DivByZero" outcome whereas the optimized right program produces an "ArithmeticOverflow". The reason is that the notion of downward safety disregards the possibility and/or presence of errors.

[4] Our experience with this example are that people keep it for artificial phantoms that do not occur in reality and which are of purely theoretical interest. Therefore, the reader is invited to compile the little C-program `main (){int x,y; x = 1/0; x = 42; printf("Catastrophe!\n")}` using the GNU-C-compiler, once as common, once with the '-O' option and watch the results.

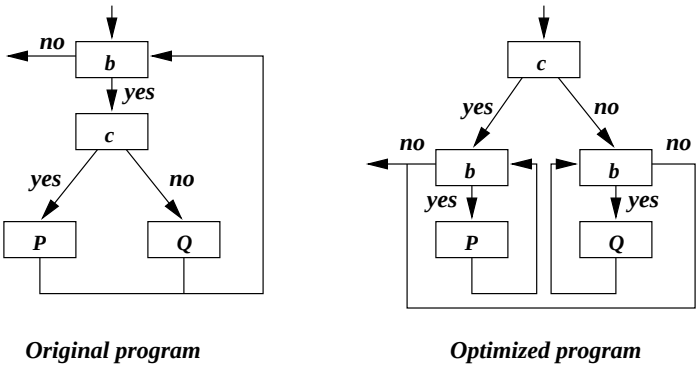

Fig. 2.3. Unswitching a loop.

Our last example for a common code optimization strategy, the so-called *unswitching* [59], is visited because it is interesting from an algebraic point of view; it utilizes the following nice distribution property. Consider a loop guarded by b where the body consists of a conditional with branches P and Q guarded by c. If both, P and Q, have no influence on c, i.e. do not change c's value, then it should be safe to move (switch) the conditional out of the loop in the sense that, according to the initial, now single, evaluation of c, only P resp. Q are iterated guarded by b. This situation is sketched in Fig. 2.3, and more technically unswitching means to replace the program

$$\mathsf{while}\ b\ \mathsf{do}\ \mathsf{if}\ c\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}\ \mathsf{od}$$

by the more efficient – though a little longer – program

$$\mathsf{if}\ c\ \mathsf{then}\ \mathsf{while}\ b\ \mathsf{do}\ P\ \mathsf{od}\ \mathsf{else}\ \mathsf{while}\ b\ \mathsf{do}\ Q\ \mathsf{od}\ \mathsf{fi}$$

if the value of c is not changed by neither P nor Q. It all seems reasonable but again suppose the evaluations of b and c yield different erroneous outcomes, e.g. an "ArithmeticOverflow" for b and a "DivByZero" for c. As above it remains questionable and a matter of taste if this transformation is admissible. But even worse, there is yet another problem concerning this transformation: It may introduce new errors into regularly terminating programs! Consider, for instance, that the given program starts in a state in which the guard b evaluates to false but the guard c to a finite error. Then the former program terminates regularly whereas the latter terminates irregularly. To make use of some vocabulary that is to be introduced soon, this transformation does not preserve total correctness and is apparently custom as, again, the presence of errors is disregarded. Notice that the classic notion of refinement is not able to deal with phenomena like these.

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

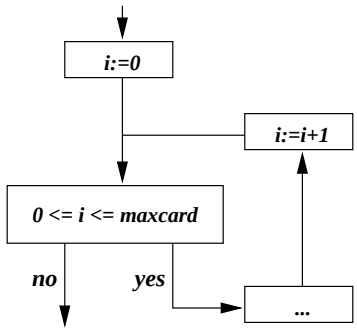

Fig. 2.4. Prototypical implementation of a for-loop.

Should optimizations be banned from verified compilers for these reasons? On account of being realistic we do not think so: We would put a spoke in our own wheel. 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 the particular application and to select just these (e.g. by means of compiler switches).

As a curiosity, we like to mention a strange example communicated by Gerhard Goos. It shows that common efficiency-improving compiler options can even lead to a translation of terminating programs into non-terminating ones in rare cases; something one really wants to avoid. The Modula-2 loop

$$\mathsf{for}\ i := 0\ \mathsf{to}\ \mathsf{maxcard}\ \mathsf{do} \ldots \ ,$$

for instance, is obviously 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 range $0 \leq i \leq \mathsf{maxcard}$; at the end of each iteration i is incremented. This is illustrated in Fig. 2.4. 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 in fact 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 a correct translation but that different applications and translation schemes preserve a different amount from the behavior of programs. In the next section we will sum up what we have learned from this little survey and lay the foundation-stone for oncoming considerations by formulating our insights in a definition of a *correct implementation*.

2.3 An Adequate Proposal

Assume again π' to be the target program which was generated from the source program π. Now suppose σ to be a possible outcome of π' started in the initial

state s, i.e. $(s, \sigma) \in R(\pi')$. If σ is also a possible result of π started in s, i.e. if σ is what we call *preserved*, everything is fine, this is the best we can expect and if this is the case for all initial states we have achieved a relational inclusion. But, as just portrayed, this might well not be the case for all those outcomes, so assume σ is *not* a possible outcome of π started in s. Well, σ could be an outcome which is impossible for π at all, e.g. it could be an error-state which does not exist on the source side. The other possibility is that σ occurs on the target side as a result of some code-optimizations which were performed during the translation (in fact, optimizations are translations). Remember that in the dead-code-elimination example the optimized program may produce any outcome, depending on the shape of P, all of which (except for "DivByZero") are impossible for the original program as it always terminates irregularly with a "DivByZero"-outcome. The strange Modula-2 example is another one: The outcome $\sigma = \infty$ is also impossible for the source program as it always terminates regularly (if the omitted body does not abort).

How do we have to deal with outcomes like these? From our point of view some of them can be treated as what we call *acceptable*. Typically, an erroneous outcome which is impossible for the source program and which occurs only as a result of a violation of some resource limitations of the target machine, e.g. finite memory or restricted arithmetic, are simply unavoidable and hence have to be accepted (apart from safety critical process programming). The other outcomes, those which are neither preserved nor accepted, let the target program behave *chaotically* or arbitrarily (one can also say that there are *chaotic* outcomes of the source program starting in s which violate its specification).

These considerations inspire us for the following approach. 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 have to be preserved literally (e.g. all regular states);
- a set AO ("*accepted outcomes*") of outcomes that may arise as a result of target program executions even if not present in the semantics of the source program (typically, these are due to a violation of some resource limitations, e.g. "StackOverflow" or "OutOfMemory"); 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).

As mentioned above, typically the regular outcomes belong to the set PO, i.e. $\Sigma \subseteq PO$, but also irregular outcomes may, e.g. "DivByZero", for debugging purposes. Moreover, if one insists on keeping all erroneous outcomes for unacceptable one naturally has the freedom to set $AO = \emptyset$.

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 each $(s, \sigma) \in R(\pi')$ (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 2.3.1 (Correct implementation). Program π' *(correctly) implements* program π w.r.t. preserved outcomes PO, accepted outcomes AO, and chaotic outcomes CO iff

$$R(\pi') \subseteq R(\pi) \cup \{(s, \sigma) \mid (\sigma \in AO) \vee (\exists \sigma' \in CO :: \ (s, \sigma') \in R(\pi))\} \ . \quad (2.4)$$

□

The notion of a correct implementation is the one that allows to reason about preservation properties of translations on a very fine-grained and detailed level if one is interested in the input-output behavior of strictly transformational programs. Moreover, it is the one that will guide and accompany us in the following. However, it is obvious that reasoning in terms of inhomogeneous relations, like $R(\pi)$, while having a correctness notion in mind that is based on a fixed but arbitrary partitioning of outcomes, like (2.4), is an ungrateful and not very elegant task. The remainder is, thus, devoted to elaborations – and applications – of more abstract, handy and manageable views on correctness which still allow, if appropriately assembled, to express fine-grained correctness properties of programs and particularly to prove implementations correct in the sense of Def. 2.3.1.

2.4 Remarks

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, 14, 33] and program refinement [5, 55, 56], each concerned with a specific correctness notion.

There is also a notion of "general correctness" (see [39], a summary of [38]) which is intended to provide a more general theory that allows reasoning about partial and total correctness (in terms of predicate transformers) as well as about demonic and angelic interpretations of nondeterminism (seen from the perspective of the underlying relational semantics). In this sense the approach presented there has the flavor of ours (Chap. 4) but it has the disadvantages that the underlying relational semantics has to satisfy certain healthiness conditions in order to be reasonable, that their understanding of predicates and, consequently, their resulting definitions are less intuitive and mostly, as common, that all kind of erroneous outcomes are identified. The "general correctness" approach thus unifies partial

and total correctness – notions that will be defined later – but leaves no space for other correctness preservation properties. Similar comments can be made for [54]. Based on a three-valued logic a family of so-called "extended predicate transformers" is derived, two of which correspond to wp resp. wlp. Thus, [54] indeed presents a unification, but the other instances of their "*ewp*" transformers are not further discussed and have no sensible meaning. Furthermore, the setup presented there also identifies finite errors and divergence. Worth mentioning in this context is the correctness notion the *Verifix* research project pursues. The observation that – for translation verification purposes – it makes essential differences whether a program delivers irregular results or no results at all and the idea to allow certain erroneous outcomes is – among others – the key to prove realistic compilers correct and to keep things manageable. The interested reader is referred to [23, 25] for a brief introduction and to [20] for a more detailed exposition.

However, we tried to motivate that these common idealizations do not lead to a realistic notion of correct implementation. On the one hand, a 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. "DivByZero" in this case) to an arbitrary outcome. On the other hand, it must be treated as acceptable, as it could also report on a limitation of the execution mechanisms at hand (e.g. standing for "OutOfMemory").

The theory typically used nowadays disregards the fact that phenomena like the ones discussed before are of theoretical *and* practical interest. 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. 4.3). It is introduced in [2] in order to justify proof rules for total correctness and is said to be not of interest in itself. On the contrary, we emphasize that relative correctness indeed is of independent interest.

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. Definition 2.3.1 serves the first part of this proposal but obviously a relational inclusion in the shape of (2.4) is hard to handle in practice. As a remedy we would like to work in a richer space than the space of relations because this would allow to reason about correctness on a more abstract level. The complete lattices of predicates (in fact this is a Boolean lattice) and predicate transformers turned out to be quite use- and helpful for these concerns, e.g. Dijkstra's wp- [17, 18] and the refinement-calculus [4, 55, 56], and the utility of the latter two mentioned theories inspires us for a sort of extension thereof. But before going into details we recall the essence of lattice theory.

3. Lattice Theory

The present thesis deals with correctness, thus it deals with semantics. There is a variety of possibilities to describe the semantics of a program, e.g. by means of transition relations on configurations which describe the behavior stepwisely or by means of relations in the shape of (2.1) which relate input states to possible outcomes. As just mentioned, an alternative is a predicate transformer semantics, e.g. by means of Dijkstra's wp- and wlp-transformers. The advantages of a semantical description like this over the ones mentioned before are manifold. Firstly, predicate transformers map post- to weakest preconditions and, thus, each predicate transformer directly corresponds to a correctness property, see Sect. 4.3, and secondly programming operators and their semantics are visually correlated which really facilitates reasoning about semantics. Furthermore, the set of monotonic predicate transformers forms a complete lattice which is a well studied framework provided with a rich and beautiful theory. Therefore, the present chapter is devoted to a brief but more or less sufficient introduction into this matter.

As common, we assume the reader to be familiar with the very basics of lattice theory. Nevertheless, on amount of being complete and to report everything that is needed to fully understand what follows, we like to recapitulate the essences. For a more profound exposition concerning lattice theory in general we recommend some of the classic text books, e.g.[10, 26]. Supporting literature for a deeper analysis of Galois connections and fixpoints are, for instance, the beautiful, comprehensible and very worth reading contributions [49, 61, 81].

3.1 Basic Definitions

Predicate Calculus. Apart from pure lattice theory we will of course make use of predicate logic and – more or less – widely known rules. This tiny paragraph is thus to be understood as an intermezzo on notational conventions and naming of quotable rules. We point out that we use a slightly different notation for quantifications which we borrow from Dijkstra's and Scholten's predicate calculus [18]. Universal quantification, for instance, of a variable x over a formula p restricted to the range r, which is traditionally denoted by, e.g., $\forall x.\ r.x \longrightarrow p.x$, will be denoted by $\forall x :\ r.x :\ p.x$. A quantification over the universe `true` can be omitted, i.e. $\forall x :\ \texttt{true} :\ p.s \iff \forall x ::\ p.x$. In analogous ways existential quantification of variable x over formula p restricted to the range r is denoted by $\exists x :\ r.x :\ p.x$ rather than the more common $\exists x.\ r.x \wedge p.x$. This kind of notation is chosen in order to dramatically increase readability because a lot of symbols,

mostly parentheses and negation signs, can be omitted and because negations of quantifications visually harmonize well. We hope the reader accepts this sparse motivation after a look at Fig. 3.1 which contains the needed predicate-logic or -calculus laws, most of which are cited from [18] as they are. Here, p, q, r and s are formulas which – unless otherwise stated – may contain the free variables x and y. We focus on universal quantification mainly as the rules of its dual are similar and a simple consequence of duality, see the first law.

[*Negation of quantification:*]
$\neg(\forall x :\ r.x :\ p.x) \ \Longleftrightarrow\ \exists x :\ r.x :\ \neg p.x$

[*Trading the range:*]
$\forall x :\ r.x \wedge q.x :\ p.x \ \Longleftrightarrow\ \forall x :\ q.x :\ \neg r.x \vee p.x$

[*Splitting the range:*]
$(\forall x :\ r.x :\ p.x) \ \wedge\ (\forall x :\ s.x :\ p.x) \ \Longleftrightarrow\ \forall x :\ r.x \vee s.x :\ p.x$

[$\forall$ *distributes over* $\wedge$*:*]
$(\forall x :\ r.x :\ p.x) \ \wedge\ (\forall x :\ r.x :\ q.x) \ \Longleftrightarrow\ \forall x :\ r.x :\ p.x \wedge q.x$

[$\forall$ *and* $\vee$] If q is independent of x, then
$q \vee \forall x :\ r.x :\ p.x \ \Longleftrightarrow\ \forall x :\ r.x :\ p.x \vee q$

[$\forall$ *distributes over* $\vee$*:*] If p and q are independent of y and x respectively, then
$(\forall x :\ r.x :\ p.x) \ \vee\ (\forall y :\ s.y :\ q.y) \ \Longleftrightarrow\ \forall x, y :\ r.x \wedge s.y :\ p.x \vee q.y$

[*Nesting and unnesting:*]
$\forall x :\ r.x :\ (\forall y :\ s.x.y :\ p.x.y) \ \Longleftrightarrow\ \forall x, y :\ r.x \wedge s.x.y :\ p.x.y$

[*Interchange of quantification:*]
$\forall x :\ r.x :\ (\forall y :\ s.y :\ p.x.y) \ \Longleftrightarrow\ \forall y :\ s.y :\ (\forall x :\ r.x :\ p.x.y)$

[$\exists$*-Introduction:*]
$\forall x, y :\ r.x.y \wedge s.y :\ p.y \ \Longleftrightarrow\ \forall y :\ (\exists x ::\ r.x.y) \wedge s.y :\ p.y$

[*Best of both worlds:*]
$(\forall x :\ r.x :\ p.x) \ \wedge\ (\exists x :\ r.x :\ q.x) \ \Longrightarrow\ \exists x :\ r.x :\ p.x \wedge q.x$

[*One point rule:*]
$\exists x :\ x = y :\ p.x \ \Longleftrightarrow\ p.y$

Table 3.1. Predicate logic laws

Lattices and partial orders. Assume given a non-empty set L equipped with two binary operations '$\cap$', called *meet*, and '$\cup$', called *join*. The triple $(L, \cap, \cup)$ is called a *lattice* if, for all $a, b, c \in L$, the following laws are satisfied.

– *Commutativity:* $a \cap b = b \cap a$ and $a \cup b = b \cup a$.
– *Associativity:* $(a \cap b) \cap c = a \cap (b \cap c)$ and $(a \cup b) \cup c = a \cup (b \cup c)$.
– *Absorption:* $a \cap (a \cup b) = a$ and $a \cup (a \cap b) = a$.

On a lattice $(L, \cap, \cup)$ a binary relation '$\subseteq$' can defined by

$$a \subseteq b \ \stackrel{\text{def}}{\Longleftrightarrow}\ a \cap b = a \ , \tag{3.1}$$

or, equivalently, $a \subseteq b \ \stackrel{\text{def}}{\Longleftrightarrow}\ a \cup b = b$. This relation satisfies the laws

– *Reflexivity:* $a \subseteq a$,
– *Antisymmetry:* if $a \subseteq b$ and $b \subseteq a$ then $a = b$, and
– *Transitivity:* if $a \subseteq b$ and $b \subseteq c$ then $a \subseteq c$

for all $a, b, c \in L$ and is thus what is known under the name of a *partial order*, in particular $(L, \subseteq)$ is a *partially ordered set*.

Suppose C is a subset of L. We call C a *chain* in L if C is non-empty and if all of its elements are related w.r.t. '$\subseteq$', i.e. if

$$C \neq \emptyset \text{ and } \forall a, b \in C :: \ a \subseteq b \vee b \subseteq a \ .$$

For an arbitrary set A an element $x \in L$ is called an *upper bound* of A if each element of A is less than or equal to x, i.e. if

$$\forall a \in A :: \ a \subseteq x \ .$$

Furthermore, an upper bound x of A is a *least upper bound* of A if it is less than or equal to any other upper bound of A, i.e. if

$$\forall y \in L : \ y \text{ is upper bound of } A : \ x \subseteq y \ .$$

By antisymmetry of '$\subseteq$' least upper bounds of A are unique, if existing at all, thus it makes sense to refer to "the" least upper bound of A which we denote by $\bigcup A$. Analogously the notion of a *lower bound* and the *greatest lower bound* are to be understood. It is an ease to show that, for all $a, b \in L$, the least upper bound $\bigcup\{a, b\}$ and the greatest lower bound $\bigcap\{a, b\}$ of a and b exist. By induction one easily proves that also $\bigcup A$ and $\bigcap A$ exist for all finite and non-empty subsets A of L.

As seen above, a lattice $(L, \cap, \cup)$ induces a partial ordered set $(L, \subseteq)$ by definition (3.1). The converse is also true. Each partially ordered set $(L, \subseteq)$ in which $\bigcup\{a, b\}$ and $\bigcap\{a, b\}$ exist for all $a, b \in L$ induces a lattice $(L, \cap, \cup)$ where the connectives '$\cap$' and '$\cup$' are defined by the identities

$$a \cap b \stackrel{\text{def}}{=} \bigcap\{a, b\} \quad \text{and} \quad a \cup b \stackrel{\text{def}}{=} \bigcup\{a, b\}$$

for $a, b \in L$. Hence, whenever talking about lattices we will freely switch between both views, the order-theoretic one, i.e. $(L, \subseteq)$, and the algebraic one, i.e. $(L, \cap, \cup)$, depending on what seems to be more appropriate in the respective situation.

For the sake of completeness – and quite obvious in set-theory – we state the following rule: For all $x, y \in L$:

$$x \subseteq y \iff \forall z : \ z \subseteq x : \ z \subseteq y \ ,$$

i.e. two elements of a lattice are related by '$\subseteq$' iff their lower bounds are suitable related, too. We refer to this property by *indirect inequality*.

A lattice $(L, \cap, \cup)$ is called *distributive* if for all $a, b, c \in L$ the identities

$$a \cap (b \cup c) \ = \ (a \cap b) \cup (a \cap c) \ \text{ and } \ a \cup (b \cap c) \ = \ (a \cup b) \cap (a \cup c)$$

hold; to be precise one of them suffices as they are equivalent.

We call a lattice $(L, \subseteq)$ *bounded* if it has a least element, typically denoted by $\bot$ ("bottom"), and a greatest element, $\top$ ("top"). If $(L, \subseteq)$ is bounded, and if for all $a \in L$ there exists an element $b \in L$ such that $a \cap b = \bot$ and $a \cup b = \top$ we call the lattice *complementary*. In this case the above b is unique and hence called *the complement* of a which we denote by $\neg a$.

A distributive, bounded and complementary lattice is a *Boolean lattice*. A widely used rule in Boolean lattices is the following: For all $a, b, c \in L$:

$$a \cap b \subseteq c \iff b \subseteq \neg a \cup c \ . \tag{3.2}$$

We will refer to (3.2) by *shunting*.

Finally, a lattice $(L, \subseteq)$ is called *complete* if $\bigcup A$ and $\bigcap A$ exist for all subsets A of L, even for infinite subsets and the empty set. Complete lattices are bounded by the smallest element $\bot = \bigcup \emptyset = \bigcap L$ and by the greatest element $\top = \bigcap \emptyset = \bigcup L$.

The standard example for a complete Boolean lattice is the powerset-lattice 2^A of an arbitrary set A. Meet and join are given by set-theoretic intersection and set-theoretic union respectively, just like the symbols '$\cap$' and '$\cup$' suggest. The order on 2^A is given by set-inclusion, and the complement $\neg B$ of an element $B \in 2^A$, i.e. a subset of A, is the set-theoretic relative complement, $A \setminus B$.

3.2 Functions between Lattices

For the remainder of this chapter we assume given some arbitrary sets A, B, C and two lattices $(L, \subseteq)$ and $(L', \subseteq')$.

Functions. The set of functions from A to B will be denoted by $(A \to B)$. The mapping in $(A \to A)$ that maps each $a \in A$ to itself is called the *identity on* A, Id_A for short. The *composition* of two functions $f \in (B \to C)$ and $g \in (A \to B)$ is a function from A to C and denoted by $f \ ; \ g$. It is, for $a \in A$, defined by $(f \ ; g)(a) = f(g(a))$. The reader might prefer – and expect – to read $f \circ g$ instead of $f \ ; \ g$ but in the framework of predicate transformers at which we finally aim the chosen notation turns out to be advantageous what readability and intuition concerns and, furthermore, has become a standard. In this sense we kindly request the reader for willingness to comprehend and to be flexible. As composition is obviously associative the triple $((A \to A), ;, Id_A)$ forms a so-called *monoid*.

The lattice of functions. The order '$\subseteq$' on L can be lifted to a binary relation '$\leq$' on $(A \to L)$, the so-called *point-wise extension of* '$\subseteq$', by defining

$$f \leq g \stackrel{\text{def}}{\iff} \forall a \in A :: \ f(a) \subseteq g(a)$$

for $f, g \in (A \to L)$. It is a short exercise to show that '$\leq$' is an order and that $((A \to L), \leq)$ is also a lattice. The meet '$\wedge$' and join '$\vee$' operations on $(A \to L)$ are lifted, too. For $f, g \in (A \to L)$ and $a \in A$ they are given by

$$(f \wedge g)(a) \stackrel{\text{def}}{=} f(a) \cap g(a) \quad \text{and} \quad (f \vee g)(a) \stackrel{\text{def}}{=} f(a) \cup g(a) \ .$$

If $(L, \subseteq)$ is distributive, bounded, Boolean or complete, then so is the lattice $((A \to L), \leq)$.

Functions from lattices to lattices. In particular, the set $(L \to L')$ of functions from the (complete) lattice $(L, \subseteq)$ to the (complete) lattice $(L', \subseteq')$ together with the pointwise extension of '$\subseteq'$' is a (complete) lattice. A function $f \in (L \to L')$ is said to be

– *monotonic* if it preserves the order, i.e. if

$$\forall a, b \in L : \ a \subseteq b : \ f(a) \subseteq' f(b) \ ,$$

and, if the lattices just mentioned are indeed complete, f is called

– *(positively) conjunctive* if it distributes over non-empty meets, i.e. if

$$f(\bigcap X) \ = \ \bigcap'\{f(x) \mid x \in X\}$$

for every *non-empty* subset X of L,

– *(positively) disjunctive* if it distributes over non-empty joins, i.e. if

$$f(\bigcup X) \ = \ \bigcup'\{f(x) \mid x \in X\}$$

for every *non-empty* subset X of L,

– *universally conjunctive* if it distributes over arbitrary meets, i.e. if

$$f(\bigcap X) \ = \ \bigcap'\{f(x) \mid x \in X\}$$

for *all* subsets X of L,

– *universally disjunctive* if it distributes over arbitrary joins, i.e. if

$$f(\bigcup X) \ = \ \bigcup'\{f(x) \mid x \in X\}$$

for *all* subsets X of L, and f is called

– *universally junctive* if f is both universally conjunctive and universally disjunctive.

Note that conjunctive and disjunctive functions are monotonic. If, e.g., f is disjunctive and $x \subseteq y$, i.e. $x \cup y = y$, then

$$f(x) \subseteq' f(x) \cup' f(y) = f(x \cup y) = f(y) \ .$$

Another well-known notion is *strictness*, in particular f is called $\bot$-*strict* if $f(\bot_L) = \bot_{L'}$ and $\top$-*strict* if $f(\top_L) = \top_{L'}$. Using this notion one would say that f is universally conjunctive (resp. disjunctive) iff f is positively conjunctive (resp. disjunctive) and $\top$-strict (resp. $\bot$-strict). The notion of *continuity* of f, known from cpo-theory, is a weaker version of disjunctivity, i.e. f distributes over arbitrary joins of chains. In lattice theory this corresponds to $\cup$-continuity and by duality there is also a notion of $\cap$-continuity. However, both notions will not be needed in the remainder. They are roughly cited for those readers familiar with them in order to stress the differences.

Galois connections. For a motivation let us have a closer look at the shunting-rule (3.2). With the notations $(a\cap) \in (L \to L)$ and $(\neg a\cup) \in (L \to L)$ mapping $b \in L$ to $a \cap b$ and $\neg a \cup b$ respectively, the shunting-rule can be re-written:

$$(a\cap)(b) \subseteq c \iff b \subseteq (\neg a\cup)(c)$$

for all $c \in L$. This kind of relation between two functions is generalized as follows. Suppose the lattices $(L, \subseteq)$ and $(L', \subseteq')$ are complete and let two functions $f \in (L \to L')$ and $g \in (L' \to L)$ be given. The pair (f, g) is called a *Galois connection* if for all $x \in L$ and $y \in L'$:

$$f(x) \subseteq' y \iff x \subseteq g(y) \ . \tag{3.3}$$

Function f is called a *lower adjoint of* g, and consequently g is called an *upper adjoint of* f. We refer to exploitations of equivalences in the shape of (3.3) also by *shunting*.

Galois connections and adjoints play an enormously helpful role whenever reasoning about complete lattices and functions between them because of the powerful properties they enjoy. Let us just collect some of them, mainly those which are of interest for our purposes. First of all we mention that lower and upper adjoints are unique so it is reasonable to refer to *the* adjoint rather than to *an* adjoint. Furthermore, the lower adjoint of a Galois connection is universally disjunctive and the upper adjoint is universally conjunctive, hence they are also monotonic. In particular, this allows to prove certain distribution properties by finding the adjoint. The converse holds, too: For each universally disjunctive function $f \in (L \to L')$ there exists a unique function $f^\sharp \in (L' \to L)$ such that $(f, f^\sharp)$ forms a Galois connection. Analogously, for each universally conjunctive function $g \in (L' \to L)$ there exists a unique function $g^\flat \in (L \to L')$ such that $(g^\flat, g)$ is also a Galois connection. Useful is also the fact that

$$f \ ; g \leq' Id_{L'} \quad \text{and} \quad Id_L \leq g \ ; f \tag{3.4}$$

hold for a Galois connection (f, g). The opposite is also true. Each pair of functions (f, g) satisfying (3.4) forms a Galois connection, i.e. (3.4) could also serve as a definition.

There is a variety of other useful facts, and we advise the reader to have a look at, e.g., [61] or [81] which present plenty of nice applications, each in its own framework aiming for very different purposes. As we will not make that heavy and tricky use of Galois connections we prefer to mention further applications whenever they are needed.

3.3 Fixpoints in Complete Lattices

The famous "Fixpoint Theorem" which is mostly attributed to Knaster and Tarski [79] and which is essentially Kleene's "Recursion Theorem" [40] (see [44] for a more far-reaching discussion concerning this) ensures that every monotonic function f on a complete lattice $(L, \subseteq)$ has a least fixpoint, μf, and a greatest fixpoint, νf. Furthermore, the set of fixpoints of f, i.e. $\mathrm{Fix}_f = \{x \in L \mid f(x) = x\}$ together with the restricted order '$\subseteq|_{\mathrm{Fix}_f}$' forms a complete lattice. In this sense the notion of a least and greatest fixpoint comes in two flavors: w.r.t. the complete lattice $(L, \subseteq)$ and w.r.t. the complete lattice $(\mathrm{Fix}_f, \subseteq|_{\mathrm{Fix}_f})$.

Proving (the foremost mentioned version of) this theorem reduces to showing that the least fixpoint μf equals the greatest lower bound of the so called *contracted elements* of f, i.e. that

$$\mu f \ = \ \bigcap \{x \in L \mid f(x) \subseteq x\} \ ,$$

resp. that the greatest fixpoint νf equals the least upper bound of the so called *expanded elements* of f, i.e. that

$$\nu f \;=\; \bigcup\{x \in L | x \subseteq f(x)\} \;\; .$$

A simple consequence of this characterization is *Park's lemma* which reads

$$\mu f \subseteq x \;\Longleftarrow\; f(x) \subseteq x \quad \text{resp.} \quad x \subseteq \nu f \;\Longleftarrow\; x \subseteq f(x) \;\; , \tag{3.5}$$

and to which we mostly refer by *induction rule*. (To be precise, this consequence is simple in the field of complete lattices and it is not for cpos, [68]). A simple application of this induction rule proves that the μ- resp. ν-operator itself is monotonic: For monotonic $f, g \in (L \to L)$,

$$\mu f \subseteq \mu g \;\Longleftarrow\; f \leq g \quad \text{and} \quad \nu f \subseteq \nu g \;\Longleftarrow\; f \leq g \;\; .$$

A further but more complicated means for proving properties concerning fixpoints of continuous functions is the *fixpoint induction principle* or *fixpoint induction* for short which has its roots in cpo-theory ([48], for instance, refers to [15] and [75]). As a complete lattice is both a cpo and a co-cpo, i.e. the cpo equipped with the reversed order, and because monotonicity turns out to be sufficient we present two stronger variations thereof, each concerned with one extreme fixpoint. (A stronger version of the least fixpoint version for cpos is an exercise in [48] and a solution can be found in, e.g., [6], the greatest fixpoint version holds by duality.) So assume $f \in (L \to L)$ to be monotonic, and let a subset P of L be given. Then $\mu f \in P$ (resp. $\nu f \in P$) provided that

1. $\forall C : \; C$ is a chain in $P : \; \bigcup C \in P$
   (resp. $\forall C : \; C$ is a chain in $P : \; \bigcap C \in P$), (P is *admissible*)
2. $\bot \in P$ (resp. $\top \in P$), and (*Base case*)
3. $\forall x \in P : \; x \subseteq f(x) : \; f(x) \in P$
   (resp. $\forall x \in P : \; f(x) \subseteq x : \; f(x) \in P$). (*Induction step*)

Sometimes it is even of interest how fixpoint properties transfer if applied to a function. Consider, for instance, $f \in (L \to L)$ and $g \in (L' \to L')$ to be monotonic. A function $h \in (L \to L')$ maps a fixpoint of f in L to a fixpoint of g in L' if the composition of h and f resp. of g and h commutes in some sense – of course the images of the involved functions have to be reasonably related – and if h additionally enjoys some particular distribution properties. To be precise one has

$$h(\mu f) \;=\; \mu g \quad \Longleftarrow \quad h \,;\, f \;=\; g \,;\, h \tag{3.6}$$

if h is universally disjunctive and analogously

$$h(\nu f) \;=\; \nu g \quad \Longleftarrow \quad h \,;\, f \;=\; g \,;\, h \tag{3.7}$$

if h is universally conjunctive. This nice rule is known under the names *transfer lemma* (e.g. in [3]) and μ- resp. *ν-fusion* (e.g. in [49]) and, e.g., [81] shows a lot of nice applications.

4. Relative Correctness

As mentioned at the end of Chap. 2 we propose more fine-grained notions of program correctness, formulated by means of predicate transformers. This chapter is dedicated to the basic definitions of what we call *relative correctness* and *relativized predicate transformers*.[1] As the names suggest there is a direct correlation between both notions and, most important, they also allow to reason about correct implementations in the sense of Def. 2.3.1 in more abstract, handy and manageable ways; just as promised in Chap. 2. But before doing so we have to introduce yet some more vocabulary and also have a more careful look at the classical treatment of program correctness and notions of translation correctness to which they give rise, because our proposal is modeled on this.

4.1 Predicates and Predicate Transformers

In Hoare-style program verification one is interested in proving programs partially or totally – notions that will be introduced soon – correct w.r.t. a pre- and a postcondition where the semantics of the latter is expressed on base of the set of regular states. Our first step is, thus, to define the concrete scenario we are going to work in as our approach can be motivated by the classical notions at best.

Just like introduced in Sect. 2.1 the set of regular states is denoted by Σ. It turns out to be convenient for our purposes to identify predicates with the set of states for which they are valid. We therefore define the set of *predicates* by

$$Pred \stackrel{\text{def}}{=} 2^{\Sigma} \ ,$$

and typically range over *Pred* by ϕ and ψ.[2] From Chap. 3 we know that the set *Pred*, as it is a powerset, together with the connectives '$\cap$' (set-theoretic intersection) and '$\cup$' (set-theoretic union) forms a complete Boolean lattice which is ordered by '$\subseteq$' (set-theoretic inclusion). As also mentioned there we denote the complement of predicate ϕ by $\neg\phi$ which equals the set-theoretic relative complement $\Sigma \setminus \phi$ of ϕ.[3] The strongest predicate (i.e. the smallest predicate because it is contained in all other predicates) is $\emptyset$, and the weakest predicate (i.e. the greatest predicate because it contains all other predicates) is Σ. To support the

[1] Again, the essence of this chapter, the roots of our proposal, are presented without proofs in [62].

[2] Readers who are not familiar with this presentation of predicates may freely switch between the following two concepts. If a predicate ϕ is understood as a mapping from the set of states to the truth values then $s \in \phi \iff \phi(s) = \mathtt{tt}$, where $\mathtt{tt}$ denotes the Boolean truth value "true".

[3] For readability reasons, and if it seems appropriate, we omit braces when considering singletons, so-called point-predicates, $\{s\}$. In particular we write $\Sigma \setminus s$ (or even $\neg s$) instead of $\Sigma \setminus \{s\}$.

intuition we denote the former also by `false` and the latter also by `true`. We also say that predicate ϕ "implies" predicate ψ if $\phi \subseteq \psi$; this reflects the view that propositional logic is a model of a Boolean lattice.

As the name suggests, a *predicate transformer* maps predicates to predicates. We restrict the set $(Pred \to Pred)$ to the monotonic ones because this makes sequential composition of predicate transformers, ';', monotonic, too. Hence, we define

$$PTrans \stackrel{\text{def}}{=} (Pred \stackrel{\text{mon.}}{\longrightarrow} Pred) \ .$$

The set of monotonic predicate transformers $PTrans$ together with the pointwise lifted connectives '$\wedge$' and '$\vee$' forms also a complete lattice (but observe that it is not a Boolean one because the complement $\neg f$ of $f \in PTrans$, defined by $\neg f.\phi = \neg(f.\phi)$, is not monotonic)[4] which is ordered by the pointwise lifted order '$\leq$'. The greatest element of the lattice $(PTrans, \leq)$ is denoted by $\top$, and the smallest element by $\bot$; they are defined, for all $\phi \in Pred$, by $\top.\phi = \texttt{true}$ and $\bot.\phi = \texttt{false}$ respectively. As mentioned before, we define the identity Id on $PTrans$ by $Id.\phi = \phi$ for all $\phi \in Pred$ such that the triple $(PTrans, ;, Id)$ forms a monoid.

Let us mention a special case that will come across later. Suppose given a function $f \in PTrans$. Then the function $(;\, f)$ gets a function $g \in PTrans$ as an argument and maps it to the composition of g and f, i.e. $(;\, f)(g) = g \ ; f$. Thus, $(;\, f) \in (PTrans \to PTrans)$ and it turns out that, roughly speaking, "$(;\, f)$ has all distribution properties" [61], in particular

$$(;\, f) \text{ is universally junctive} \ . \tag{4.1}$$

4.2 The Classical Setup

The classic literature on Hoare-style program verification and the refinement calculus identifies, for the sake of simplicity (and notational beauty), divergence and failure outcomes or even fully ignores failures. In our setting this amounts to assuming that Ω contains just one symbol, say '$\ddagger$',[5] which represents any kind of irregular outcomes, as there are divergence and finite errors. In this case, the relational semantics $R(\pi)$ is a subset of $\Sigma \times (\Sigma \cup \{\ddagger\})$. 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 so suppose given an arbitrary set Ω of irregular outcomes. The definitions of total and partial correctness below treat all irregular outcomes as if they were identified and can thus be equivalently read in both models (unless otherwise mentioned).

[4] Yet some more 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 common $f(x)$. Moreover, we adopt the usual convention that function application associates to the left, i.e. $f.x.y$ means $(f.x).y$.

[5] The symbol '$\bot$' is reserved.

All oncoming results of the next section are presented without proof because firstly they are more or less obvious and secondly they will soon become simple corollaries of more general results presented in Sect. 4.3 and Chap. 5.

4.2.1 Partial and total correctness

Let us directly come to a first definition which readers familiar with program verification will hopefully accept right away.

Definition 4.2.1 (Partial Correctness). Program π is called *partially correct* w.r.t. a precondition $\phi \in \mathit{Pred}$ and postcondition $\psi \in \mathit{Pred}$, denoted by $\{\phi\}\pi\{\psi\}$ for short, if

$$\forall s, \sigma : \ s \in \phi \wedge (s, \sigma) \in R(\pi) : \ \sigma \in \psi \cup \Omega \ .$$

□

Intuitively, program π is partially correct w.r.t. ϕ and ψ if each *regularly* terminating computation starting in an initial state satisfying ϕ results in a state satisfying ψ. Note how the restriction to regular results is expressed by allowing all outcomes in Ω.

This is one extreme point of view because we pay attention to regular results only, all irregular outcomes are don't-cares in this sense. The other extreme concept is to demand that π delivers only regular results which leads to the following definition.[6]

Definition 4.2.2 (Total Correctness). Program π is said to be *totally correct* w.r.t. precondition $\phi \in \mathit{Pred}$ and postcondition $\psi \in \mathit{Pred}$, denoted by $[\phi]\pi[\psi]$ for short, if

$$\forall s, \sigma : \ s \in \phi \wedge (s, \sigma) \in R(\pi) : \ \sigma \in \psi \ .$$

□

Again, the requirement that π delivers no irregular outcome is nicely expressed by allowing only outcomes contained in the postcondition which is a set of regular states (or in other words by *not* allowing outcomes contained in Ω).

An elegant way of expressing partial and total correctness is by means of monotonic predicate transformers, i.e. monotonic mappings on the space of predicates. Dijkstra [17, 18] considers two predicate transformers, the *weakest liberal precondition* transformer wlp is suited to partial correctness and the *weakest precondition* transformer wp to total correctness. (Note that wlp and wp itself are simply

[6] This is the place to resume the remarks from Sect. 2.1. The very classical notion of total correctness demands that π always delivers a regular result satisfying the postcondition whenever started in an initial state satisfying the precondition. This parlance particularly implies that π delivers an outcome at all if started in such an initial state. Note that, in our scenario, a relational semantics need not to be total so we could require totality of the underlying relational semantics here in order to match the classical notion. However, we refrain from doing so as this would be a far too strong restriction without bringing any advantages apart from the notional accordance but involving a lot of notational circumstances. The reader is kindly requested to accept the notion of "total correctness" as defined here and we will be quite careful to indicate properties of the relational semantics if they are relevant.

transformers, they become predicate transformers if applied to a program. This becomes clearer after the definitions below.)

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

Definition 4.2.3 (wlp and wp). The *weakest liberal precondition predicate transformer* of program π, $\mathsf{wlp}.\pi \in PTrans$, is given by

$$\mathsf{wlp}.\pi.\psi \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall \sigma : \ (s,\sigma) \in R(\pi) : \ \sigma \in \psi \cup \Omega\}$$

and the *weakest precondition predicate transformer*, $\mathsf{wp}.\pi \in PTrans$, is defined as

$$\mathsf{wp}.\pi.\psi \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall \sigma : (s,\sigma) \in R(\pi) : \ \sigma \in \psi\} \ .$$

□

As the name suggests, $\mathsf{wlp}.\pi.\psi$ (resp. $\mathsf{wp}.\pi.\psi$) is the weakest predicate ϕ satisfying the Hoare-triple $\{\phi\}\pi\{\psi\}$ (resp. $[\phi]\pi[\psi]$). Let us show that "weakest" is indeed a proper attribute such that the below equivalences could also serve as a definition of wp and wlp.

Lemma 4.2.1 ($\mathsf{wlp}.\pi$ and $\mathsf{wp}.\pi$ *are* weakest transformers).

$$\begin{array}{lcl} \phi \subseteq \mathsf{wlp}.\pi.\psi & \Longleftrightarrow & \{\phi\}\pi\{\psi\} \ , \text{and} \\ \phi \subseteq \mathsf{wp}.\pi.\psi & \Longleftrightarrow & [\phi]\pi[\psi] \ . \end{array}$$

□

The transformers $\mathsf{wlp}.\pi$ and $\mathsf{wp}.\pi$ provide abstractions of $R(\pi)$ suited to partial and total correctness respectively. Both carry less information than $R(\pi)$ itself. This can be seen from the following examples in which we again use '|' to denote (demonic) nondeterministic choice (see p. 12).

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

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

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

$\mathsf{wp}.\pi$ 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 $|\Omega| = 1$, the relational semantics $R(\pi)$ can be reconstructed from $\mathsf{wp}.\pi$ *together* with $\mathsf{wlp}.\pi$.

Lemma 4.2.2 (Reconstructing $R(\pi)$ from wlp and wp). For $\Omega = \{\ddagger\}$:

$$R(\pi) \ = \ \{(s,s') \mid s \notin \mathsf{wlp}.\pi.\neg s'\} \cup \{(s,\ddagger) \mid s \notin \mathsf{wp}.\pi.\mathtt{true}\} \ .$$

□

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 because they either ignore or identify all kinds of erroneous outcomes. The proposal below, however, allows a reconstruction of the relational semantics even in the more realistic world by a greater selectivity, i.e. by a more subtle differentiation between specific outcomes (cf. Lemma 4.3.2).

4.2.2 Implementation correctness

There are three natural ways to approach translation correctness. Firstly, one can focus on *properties* that transfer from source to target programs. This point of view is particularly adequate if one is interested in program verification mainly.[7] Secondly, one might focus on the *outcomes* produced by the source and target program. This is the adequate way 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 and it is the most manageable view. 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 first, 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, e.g. the pre- and postcondition, transfers from π to π'. Two natural notions of this kind are preservation of partial and preservation of total correctness.

Definition 4.2.4 (Preservation of partial and total correctness).

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

□

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

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 Chap. 2 and we resort in the theorem below to the notion of a correct implementation introduced in Def. 2.3.1. The theorem shows that we can interpret PPC and PTC also in terms of outcomes in a natural way.

[7] Concentrating on and dealing with preservation of properties was the medium the ProCoS project [12, 13] aimed at. Its follow-up, the *Verifix* project [23], acts similarly to prove realistic compilers correct. Though this view on translation correctness is widely common the notions of preservation of partial correctness and preservation of total correctness can be attributed to those research groups.

Theorem 4.2.1 (Outcome interpretation of PPC and PTC).

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

□

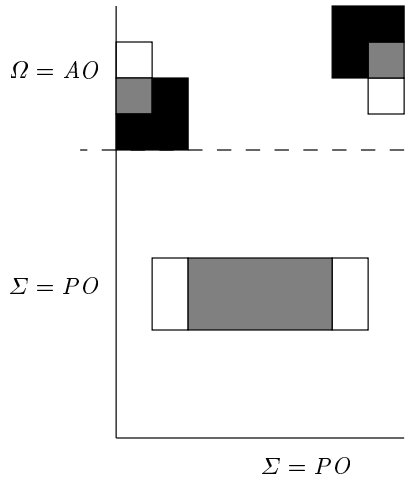

Fig. 4.1. Preservation of partial correctness, relational view.

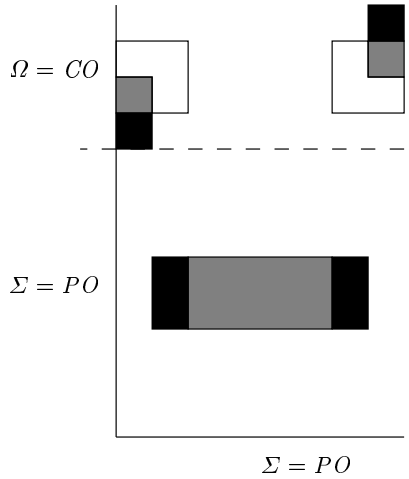

Fig. 4.2. Preservation of total correctness, relational view.

To illustrate the outcome interpretation one can draw some pictures like the ones in Figs. 4.1 and 4.2. There, the white boxes belong to the relational semantics of the source program π, the black boxes to the relational semantics of the target program π', and the grey boxes represent pairs both semantics have in common. Let us discuss the easier case, preservation of partial correctness, first. Verbalizing

Def. 2.3.1, each outcome produced by the target program is either an outcome that is also possible for the source program or is just any irregular outcome. This expresses in choosing $AO = \Omega$ and $CO = \emptyset$ and is visualized in Fig. 4.1. The second case, preservation of total correctness, is less intuitive. Here, Theorem 4.2.1 tells us to choose $AO = \emptyset$ and $CO = \Omega$. Then the situation sketched in Fig. 4.2 shows a correct implementation w.r.t. this choice because each computation of the target program that starts in an initial state, say s, delivers either an outcome which is also possible for the source program or just any (regular or irregular) outcome if s is an initial state for which the source program behaves chaotically. Notice that the target program delivers regular results for some initial states where the source program yields erroneous outcomes, as all erroneous outcomes are treated as chaotically this does not harm. In both cases, PPC and PTC, we observe a relational inclusion on the level of regular states, for the irregular states on the other side there is no restriction. In this sense, PPC and PTC are transitive notions because relational inclusion is transitive. Note furthermore that, for simplicity and readability, the pictures sketched here represent total programs which produce either regular or irregular outcomes but they straightforwardly transfer to the general case.

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 order: π' should implement π iff its interpretation in the model is related to the interpretation of π by the order.

For PPC and PTC adequate interpretations are well-known: They are given by wlp and wp.

Theorem 4.2.2 (Refinement characterization of PPC and PTC).

1. Program π' implements program π w.r.t. PPC iff $\mathsf{wlp}.\pi \leq \mathsf{wlp}.\pi'$.
2. Program π' implements program 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, i.e. $R(\pi') \subseteq R(\pi)$, can be regained from wlp and wp. In this case,

$$R(\pi') \subseteq R(\pi) \quad \text{iff} \quad \mathsf{wlp}.\pi \leq \mathsf{wlp}.\pi' \text{ and } \mathsf{wp}.\pi \leq \mathsf{wp}.\pi' \ . \tag{4.2}$$

Again, this is no longer true if $|\Omega| > 1$ but our approach below presents a remedy, cf. Corollary 4.3.2.

It follows from (4.2) that for the examples discussed in Chap. 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 bad 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 and finally, establish a finer framework in the next section.

4.3 The Relativized Setup

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, i.e. the irregular outcomes that are not accepted are taken as disproof. Colloquially speaking, we are not only interested in the border cases, i.e. accepting all or no irregular outcomes, but we will vary between those two frontiers by accepting just some of them.

4.3.1 Relative correctness

Suppose given a set $A \subseteq \Omega$ of outcomes to be accepted. After a closer look at definitions 4.2.1 and 4.2.2 the following definition seems proximate, just replace Ω resp. $\emptyset$ by A.

Definition 4.3.1 (Relative Correctness). A program π is called *relatively correct* w.r.t. a precondition $\phi \in Pred$, a postcondition $\psi \in Pred$, and a set A of accepted outcomes, denoted by $\langle\phi\rangle\pi\langle\psi\rangle_A$ for short, if

$$\forall s, \sigma : \; s \in \phi \wedge (s, \sigma) \in R(\pi) : \; \sigma \in \psi \cup A \;\; .$$

□

Intuitively, a program π is relatively correct[8] if the following holds.

> Whenever π is started in a state satisfying ϕ and delivers an outcome at all we can be sure that either π terminates regularly in a state satisfying ψ, irregularly with a failure in A, or, if $\infty \in A$, diverges.

It is important to notice that relative correctness is not one single notion but a family, each member of which is parameterized by one particular set A of accepted outcomes.

We can also define a corresponding predicate transformer along the lines of wlp and wp. It is called the *weakest relative precondition* transformer wrp_A.[9]

Definition 4.3.2 (wrp). The *weakest relative precondition predicate transformer* of program π w.r.t. a set A of outcomes to be accepted, $\mathsf{wrp}_A.\pi \in PTrans$, is defined by

[8] Of course, "correctness" itself is a relative notion as it has a meaning only w.r.t. a specification, e.g. by means of a pre- and a post-condition. However, we chose these words because relative correctness is it even more: relatively w.r.t. some erroneous outcomes to be accepted.

[9] 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 would, for instance, lead to a more complicated definition of sequential composition of predicate transformers. In fact, some expected rules would not hold at all.

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

□

Again, there is the following equivalence showing that $\mathsf{wrp}_A.\pi$ indeed deserves the name "weakest" relative precondition predicate transformer and by this means it proves Lemma 4.2.1. We refrain from a proof of Lemma 4.3.1 here as it becomes a corollary of Lemma 5.2.1 in Chap. 5.

Lemma 4.3.1 ($\mathsf{wrp}.\pi$ *is* the weakest transformer).

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

□

These relativized notions generalize the classical ones. It is rather obvious that partial and total correctness are just the border cases of relative correctness for the sets $A = \Omega$ and $A = \emptyset$. Consequently, wlp and wp are the two extreme instances of wrp_A.

Corollary 4.3.1 (Extreme instances of A).

1. $\{\phi\}\pi\{\psi\} \iff \langle\phi\rangle\pi\langle\psi\rangle_\Omega$ and $[\phi]\pi[\psi] \iff \langle\phi\rangle\pi\langle\psi\rangle_\emptyset$
2. $\mathsf{wlp}.\pi = \mathsf{wrp}_\Omega.\pi$ and $\mathsf{wp}.\pi = \mathsf{wrp}_\emptyset.\pi$

□

To follow the lines from the last section we continue with the preservation properties of relative correctness.

4.3.2 Implementation correctness

Each set $A \subseteq \Omega$ gives rise to a notion of translation correctness w.r.t. 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 of a correct implementation as introduced in Def. 2.3.1.

Theorem 4.3.1 (Preservation of relative correctness). The following three characterizations are equivalent.

1. (Preservation) $\forall \phi, \psi : \langle\phi\rangle\pi\langle\psi\rangle_A : \langle\phi\rangle\pi'\langle\psi\rangle_A$.
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 $\Omega \setminus A$.

Proof. We start with the equivalence of 2 and 3:

$$\begin{array}{ll} & \mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi' \\ \iff & \quad \{\text{Lifted order and definition of } \mathsf{wrp}_A\} \\ & \forall \psi, s : (\forall \sigma : (s,\sigma) \in R(\pi) : \sigma \in \psi \cup A) : \\ & \qquad (\forall \sigma : (s,\sigma) \in R(\pi') : \sigma \in \psi \cup A) : \\ \iff & \quad \{\text{Nesting and renaming}\} \end{array}$$

$$
\begin{array}{ll}
 & \forall \psi, s, \sigma :\ (\forall \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \in \psi \cup A)\ \wedge\ (s,\sigma) \in R(\pi') : \\
 & \qquad \sigma \in \psi \cup A \\
\Longleftrightarrow & \quad \{\text{Trading the range and negation of } \forall\} \\
 & \forall \psi, s, \sigma :\ (s,\sigma) \in R(\pi') : \\
 & \qquad \sigma \in \psi \cup A\ \vee\ (\exists \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \notin \psi \cup A) \\
\Longleftrightarrow & \quad \{\exists \text{ distributes over } \vee, \text{ resolve negation}\} \\
 & \forall \psi, s, \sigma :\ (s,\sigma) \in R(\pi') : \\
 & \qquad \sigma \in \psi \cup A\ \vee \\
 & \qquad \exists \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \in \neg\psi\ \vee \\
 & \qquad \exists \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \in \Omega \setminus A \\
\Longleftrightarrow & \quad \{\text{For ``}\Longrightarrow\text{'' choose } \psi = \{s' \in \Sigma \mid (s,s') \in R(\pi)\}, \\
 & \qquad \text{in ``}\Longleftarrow\text{'' take } \sigma \text{ for the searched } \sigma'\} \\
 & \forall s, \sigma :\ (s,\sigma) \in R(\pi') : \\
 & \qquad (s,\sigma) \in R(\pi)\ \vee \\
 & \qquad \sigma \in A\ \vee \\
 & \qquad \exists \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \in \Omega \setminus A \\
\Longleftrightarrow & \quad \{\text{Relational inclusion}\} \\
 & R(\pi') \subseteq R(\pi)\ \cup \\
 & \qquad \{(s,\sigma) \mid (\sigma \in A) \vee (\exists \sigma' \in \Omega \setminus A ::\ (s,\sigma') \in R(\pi))\}\ .
\end{array}
$$

Of course, the equivalence of 1 and 2 deserves a proof, too, but we like to put off the reader to Chap. 5. There, the underlying state spaces are, on contrary to this chapter, not assumed to be equal which is a quite more realistic assumption. For a motivation, however, the homogeneous case, as presented here, is appropriate. Theorem 5.4.1 is concerned with a respective statement in the inhomogeneous case and, thus, Theorem 4.3.1 is a simple consequence. The equivalence of 2 and 3 is also considered there but it does – at least visually – not easily transfer to the present scenario so we decided to state the simpler proof here.
□

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 $\Omega \setminus A$ because the source program is not relatively correct and consequently the target program needs neither. On the other hand 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.

Note that Theorem 4.3.1 above particularly proves Theorems 4.2.1 and 4.2.2 of the classical setup. Since all three statements above are equivalent, each of them could serve as a definition of a notion like *preservation of relative correctness (PRC) w.r.t. A* and we leave it to the reader which is the most preferable one. If this notion is used in the remainder we refer to Theorem 4.3.1.

As done for the border cases, PPC and PTC, we illustrate the outcome interpretation of preservation of relative correctness. Fig. 4.3 shows a correct imple-

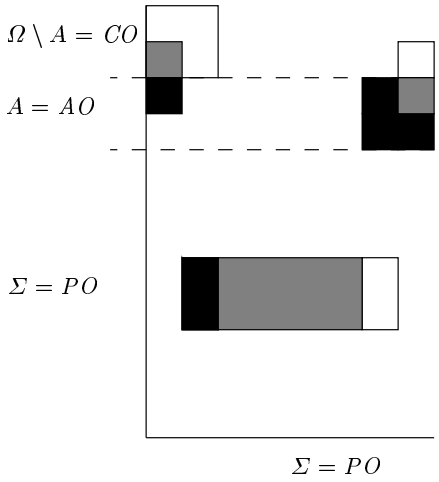

Fig. 4.3. Preservation of relative correctness, relational view.

mentation w.r.t. $AO = A$, $CO = \Omega \setminus A$ and $PO = \Sigma$ because each computation of the target program starting in an initial state, say s, delivers either an outcome that is also possible for the source program (the grey boxes), an outcome that is accepted (the upper black boxes) or just any outcome if s is an initial state for which the source program behaves chaotically (the lower black box). It is noteworthy that there is no relational inclusion, but anyhow preservation of relative correctness is a transitive notion in the sense mentioned in the discussion of Figs. 4.1 and 4.2; Chap. 5 deals with observations like these.

This looks all fine, but it is not as general as the aspired notion of a correct implementation, Def. 2.3.1, 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 the index of wrp is just accepted, not preserved; and the outcomes in $\Omega \setminus A$ are treated chaotically. What about those failure outcomes we really want to preserve? A compiler user, for instance, might require that an observed outcome "DivByZero" 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, say, full control over those outcomes and can prove that they are preserved. More formally, we have the following result.

Theorem 4.3.2 (PRC vs. implementation correctness). If $\Sigma \subseteq PO$ then π' implements π w.r.t. preserved outcomes PO, accepted outcomes AO, and chaotic outcomes CO iff

$$\forall A : \ AO \subseteq A \subseteq AO \cup (PO \cap \Omega) : \ \mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi' \ .$$

Proof. Assume $\Sigma \subseteq PO$, in particular this means $AO, CO \subseteq \Omega$. We start to shuffle the formula on the right hand side,

$$\begin{array}{ll} & \forall A : \ AO \subseteq A \subseteq AO \cup (PO \cap \Omega) : \ \mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi' \\ \Longleftrightarrow & \quad \{\text{Theorem 4.3.1 above and relational inclusion}\} \end{array}$$

$$
\begin{array}{l}
\forall A:\ AO \subseteq A \subseteq AO \cup (PO \cap \Omega):\\
\qquad \forall s,\sigma:\ (s,\sigma) \in R(\pi'):\\
\qquad\qquad (s,\sigma) \in R(\pi)\ \vee\ \sigma \in A\ \vee\\
\qquad\qquad (\exists \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in \Omega \setminus A)\\
\Longleftrightarrow \qquad \{\text{Interchange of quantification}\}\\
\forall s,\sigma:\ (s,\sigma) \in R(\pi'):\\
\qquad \forall A:\ AO \subseteq A \subseteq AO \cup (PO \cap \Omega):\\
\qquad\qquad (s,\sigma) \in R(\pi)\ \vee\ \sigma \in A\ \vee\\
\qquad\qquad (\exists \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in \Omega \setminus A)\ ,
\end{array}
$$

and it is this formula we are going to show to be equivalent to

$$
\begin{array}{l}
\forall s,\sigma:\ (s,\sigma) \in R(\pi'):\\
\qquad (s,\sigma) \in R(\pi)\ \vee\ \sigma \in AO\ \vee\ (\exists \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in CO)\ ,
\end{array}
$$

which itself is equivalent to the claim on the left hand side. The proof relies on a similar "trick", an appropriate instantiation, as in the previous proof; because this time it is even more "tricky" we perform a point-wise proof.

"$\Longleftarrow$" Let the right hand side, i.e. the foremost formula, hold and take some arbitrary s and σ such that $(s,\sigma) \in R(\pi')$, $(s,\sigma) \notin R(\pi)$ and $\sigma \notin AO$. It remains to show that

$$\exists \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in CO\ ,$$

so let us assume the opposite, i.e. assume

$$(*) \qquad \forall \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in PO \cup AO\ .$$

It follows from the premise by taking $A = AO$ that $(s,\sigma) \in R(\pi)$, $\sigma \in AO$ or

$$\exists \sigma_0:\ (s,\sigma_0) \in R(\pi):\ \sigma_0 \in (\Omega \cap PO) \cup (\Omega \cap CO)\ ,$$

By the choice of s and σ and the assumption $(*)$ it follows from the "best of both worlds" law that

$$\exists \sigma_0:\ (s,\sigma_0) \in R(\pi):\ \sigma_0 \in \Omega \cap PO\ .$$

We collect all those σ_0 in

$$S \stackrel{\text{def}}{=} \{\sigma_0 \mid (s,\sigma_0) \in R(\pi) \wedge \sigma_0 \in \Omega \cap PO\}$$

and have a closer look at $A_S \stackrel{\text{def}}{=} AO \cup S$. Obviously A_S satisfies the premise $AO \subseteq A_S$ and $A_S \subseteq AO \cup (PO \cap \Omega)$, hence we conclude that

$$(s,\sigma) \in R(\pi)\ \vee\ \sigma \in A_S\ \vee\ (\exists \sigma':\ (s,\sigma') \in R(\pi):\ \sigma' \in \Omega \setminus A_S)\ .$$

Well, we know that $(s,\sigma) \notin R(\pi)$, and if $\sigma \in A_S$ then $\sigma \in AO$ or $(s,\sigma) \in R(\pi)$ in contradiction to the choice of σ. Therefore we conclude the existence of σ' with $(s,\sigma') \in R(\pi)$ and $\sigma' \in \Omega \setminus A_S$. But this means that $(s,\sigma') \in R(\pi)$, $\sigma' \in \Omega$, $\sigma' \notin AO$ and $\sigma' \notin \Omega \cap PO$. Thus, $\sigma' \in \Omega \cap \overline{AO} \cap \overline{\Omega \cap PO} = CO$ because $CO \subseteq \Omega$, and this contradicts our assumption $(*)$.

"$\Longrightarrow$" is a little simpler. We assume the left hand side, i.e. the latter formula above, to hold and choose s, σ and an A with $AO \subseteq A \subseteq AO \cup (PO \cap \Omega)$ arbitrarily such that $(s,\sigma) \in R(\pi')$, $(s,\sigma) \notin R(\pi)$ and $\sigma \notin A$. We are left with showing the existence of a σ_0 with $(s,\sigma_0) \in R(\pi)$ and $\sigma_0 \in \Omega \setminus A$. We have a look at our premise and observe firstly that $(s,\sigma) \notin R(\pi)$ and secondly that $\sigma \notin AO$ because otherwise $\sigma \in A$ follows in contradicting the choice of σ. Thus,

$$\exists \sigma' :\ (s,\sigma') \in R(\pi) :\ \sigma' \in CO$$

holds. Now, by the choice of A,

$$A \subseteq AO \cup (PO \cap \Omega) \implies A \subseteq AO \cup PO \iff CO \subseteq \overline{A}$$

and because $CO \subseteq \Omega$ we conclude that $CO \subseteq \Omega \cap \overline{A} = \Omega \setminus A$. Choosing this σ' for the searched σ_0 completes the proof.
□

As seen in the proof, the premise $\Sigma \subseteq PO$ is essential. We have thus not completely solved our exercise to express the most general case of Def. 2.3.1 by means of predicate transformers but we think the requirement to preserve all regular outcomes is barely limiting but rather natural.

As a corollary to Theorem 4.3.2, 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. In particular this proves (4.2).

Corollary 4.3.2 (Regain relational inclusion from wrp).

$$R(\pi') \subseteq R(\pi) \iff \forall A \subseteq \Omega :: \mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi'$$

□

Relativized refinement enables us, thus, to be as fine-grained w.r.t. outcomes as on the relational level, if desired. In particular we can completely regain the relational semantics – which was the base of the definition – from the predicate transformer level. Note that, finally, this proves also Lemma 4.2.2 of the classical setup, Sect. 4.2.

Lemma 4.3.2 (Reconstructing $R(\pi)$ from wrp).

$$R(\pi) \ = \ \{(s,s') \mid s \notin \mathsf{wrp}_\Omega.\pi.\neg s'\} \ \cup \ \{(s,\omega) \mid s \notin \mathsf{wrp}_{\Omega\setminus\{\omega\}}.\pi.\mathtt{true}\}$$

Proof. We distinguish two cases: for regular outcomes s'

$$\begin{array}{ll} & s \notin \mathsf{wrp}_\Omega.\pi.\neg s' \\ \iff & \quad \{\text{Definition of wrp and negation}\} \\ & \exists \sigma :\ (s,\sigma) \in R(\pi) :\ \sigma \notin (\Sigma \setminus s') \cup \Omega \\ \iff & \quad \{\text{Complementation w.r.t. } \Sigma \cup \Omega\} \\ & \exists \sigma :\ (s,\sigma) \in R(\pi) :\ \sigma \in \overline{(\Sigma \setminus s') \cup \Omega} \\ \iff & \quad \{\overline{(\Sigma \setminus s') \cup \Omega} = \{s'\} \text{ because } s' \in \Sigma\} \end{array}$$

$$\begin{array}{ll} & \exists\sigma :\ (s,\sigma) \in R(\pi) :\ \sigma = s' \\ \Longleftrightarrow & \qquad \{\text{One point rule}\} \\ & (s,s') \in R(\pi)\ , \end{array}$$

and for irregular outcomes ω, the proof of which is omitted due to analogy.
□

In summary, although the notion of a correct implementation from Def. 2.3.1 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 some As. Verbalized differently, to prove that a translation preserves a particular relative correctness property, given by means of a particular set A, it suffices to show one single refinement. But proving refinement for a family of wrp_A-transformers, i.e. for some As, may also be of interest, namely to prove that a translation is correct in the sense of Def. 2.3.1; note that this is not a relative correctness property because also erroneous outcomes are preserved in this case but the relative correctness notion only accepts such or keeps them for chaotic.

This, among others, is a main achievement because Theorem 4.3.2 serves the second part of our proposal sketched at the end of Chap. 2: Besides our family of predicate transformers being quite comprehensible and manageable – due to the rich framework of complete lattices they work in – they also allow to prove implementations to be correct in the sense of Def. 2.3.1. We thus hope the reader accepts and appreciates our concept of parameterizing assertions w.r.t. accepted outcomes because this is the basic step in keeping things practicable.

5. Inhomogeneity and Compositionality

The major motivation for the introduction of the wrp-family was to be more realistic than ordinary (there is more than just regularity and divergence in practice) while trying to preserve the benefits of the idealized setting (separating outcomes into ones to be accepted and others to be rejected allows similar calculations in the same framework).

However, reality is even more vicious. Of course, when reasoning about translation correctness it is still unrealistic to assume source and target programs to operate on a common state space. This assumption is quite reasonable for validating code optimizations (typically, the code is just re-arranged after or before the actual translation) but translating programs of a source- to programs of a target-language characteristically includes a change of the state space. A numerical (source-) program, for instance, might use the natural numbers as data whereas its executable typically uses bit-representations thereof (some computer algebra systems are an exception). Bad as it could be, even the state space representing the input and the state space representing the output of one and the same program may differ. An obvious example is the compiler itself which translates source to target programs such that input and output data are of different type; e.g. the former is a text file whereas the latter is an executable binary. We will see in a moment how to handle this kind of inhomogeneity in the predicate transformer setting, in particular how wrp-based reasoning gets along with changes of the involved state spaces.

Furthermore, we like to outline the following scenarios which are characteristic for the development of commercial compilers nowadays. Typically, a compiler is not built monolithically but in smaller and thus more handy steps. As a direct translation from source data, e.g. simple program text, to target data, e.g. binaries, might be a far too giant step, several so-called intermediate languages are introduced such that the actual translation is separated into hopefully more manageable and easier tasks. The arising question is which correctness preservation properties the composed compiler – the one translating source to target programs – inherits from the components it consists of. In other words, how do correctness preservation properties transfer if compilers are composed *vertically*? Now consider two compilers, each of which preserves some kind of correctness. What happens if the second is fed with the output of the first; worded differently, what about the compiler that consists of this two given compilers composed *horizontally* or, better known, sequentially? Besides this general questions one has to be rather careful when composing compilers resp. when reasoning about correctness prop-

erties of that compound. Typically, the modules are built separately from each other and it might be the case that the state space of the target language of the, say, ith translation does not exactly match the state space of the source language of the $(i+1)$th translation. Furthermore, each group working on a specific part of the translation might have a slightly different understanding of program semantics or how source and target language data is related. In cases like these one has to respect certain common states covered by both languages and the involved programs resp. data representations have to be sensibly related.

Besides a discussion from the wrp-transformer viewpoint these scenarios are also surveyed from a relational perspective for the following reasons. Firstly it is hard to convince practitioners to think in terms of predicate transformers because they keep the predicate transformer setting for much too abstract and too idealized. It thus seems appropriate to translate the results to the relational setting in order to stress the elegance of wrp-based reasoning once more. Secondly we will also get to know one of the rare advantages of pointwise reasoning in terms of relations. Though all presented results are valid in both worlds, the world of predicate transformers and the world of relations, it turns out that seeking out *weak* requirements for their validity is more natural and obvious in a pointwise fashion. However, this is not very surprising as the sketched situations are indeed very queer and inhomogeneous and because pointwise reasoning is obviously more precise than abstract reasoning on the predicate level.

5.1 Data Representations and their Composition

Consider two state spaces, say Σ_{SL} and Σ_{TL}, which – as their names suggest – are meant to represent the data, the set of regular states, of some source and target language respectively. Of course, the states of source and target language are expected to be somehow related; e.g., for each state of Σ_{SL} one has none, one or even more than one state belonging to Σ_{TL}, each of which is kept for being just a more concrete representation of the more abstract state on the source level. (Again, consider a restricted n-bit representation for the natural numbers.) Thus, we call a relation

$$\rho_{\mathrm{SL,TL}} \subseteq \Sigma_{\mathrm{SL}} \times \Sigma_{\mathrm{TL}} \tag{5.1}$$

a *data representation relation*[1]. In the field of predicate transformers one is endeavored to avoid a pointwise reasoning. Therefore, we are more interested in the images of set under a data representation relation, in other words we are interested in induced predicate transformers because subsets of regular states were introduced under the name "predicates". The *(right-) image* $F_{\rho_{\mathrm{SL,TL}}}$ of $\rho_{\mathrm{SL,TL}}$ is a function of type $2^{\Sigma_{\mathrm{SL}}} \to 2^{\Sigma_{\mathrm{TL}}}$, and is, for $\phi \in 2^{\Sigma_{\mathrm{SL}}}$, defined by[2]

[1] A data representation relation plays the role of a *coupling invariant* (see, e.g., [61]) because it can be kept for a predicate which couples both state spaces by describing which properties remain invariant under the respective viewpoint.

[2] This function is just the image of ϕ under $\rho_{\mathrm{SL,TL}}$ which is usually denoted by $\rho_{\mathrm{SL,TL}}(\phi)$; it becomes clear soon why an own function-name is preferable. We will, however, use the notation $\rho_{\mathrm{SL,TL}}(\phi)$ as a substitute for $F_{\rho_{\mathrm{SL,TL}}}(\phi)$ whenever this seems appropriate.

$$F_{\rho_{\mathrm{SL,TL}}}(\phi) \stackrel{\mathrm{def}}{=} \{t \in \Sigma_{\mathrm{TL}} \mid \exists s \in \Sigma_{\mathrm{SL}} :: (s,t) \in \rho_{\mathrm{SL,TL}} \wedge s \in \phi\} \ . \tag{5.2}$$

Let us briefly have a closer look at this predicate transformer. For $\psi \subseteq \Sigma_{\mathrm{TL}}$ we calculate

$$\begin{array}{ll}
 & F_{\rho_{\mathrm{SL,TL}}}(\phi) \subseteq \psi \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\} \\
 & \forall t \in \Sigma_{\mathrm{TL}} : \ t \in F_{\rho_{\mathrm{SL,TL}}}(\phi) : \ t \in \psi \\
\Longleftrightarrow & \quad \{\text{Definition of } F_{\rho_{\mathrm{SL,TL}}}\} \\
 & \forall t \in \Sigma_{\mathrm{TL}} : \ (\exists s \in \Sigma_{\mathrm{SL}} :: \ (s,t) \in \rho_{\mathrm{SL,TL}} \wedge s \in \phi) : \ t \in \psi \\
\Longleftrightarrow & \quad \{\exists\text{-Introduction}\} \\
 & \forall t \in \Sigma_{\mathrm{TL}}, s \in \Sigma_{\mathrm{SL}} : \ (s,t) \in \rho_{\mathrm{SL,TL}} \wedge s \in \phi : \ t \in \psi \\
\Longleftrightarrow & \quad \{\text{Unnesting}\} \\
 & \forall s \in \Sigma_{\mathrm{SL}} : \ s \in \phi : \ (\forall t \in \Sigma_{\mathrm{TL}} : \ (s,t) \in \rho_{\mathrm{SL,TL}} : \ t \in \psi) \\
\Longleftrightarrow & \quad \{\text{Define } G_{\rho_{\mathrm{SL,TL}}} \in (2^{\Sigma_{\mathrm{TL}}} \to 2^{\Sigma_{\mathrm{SL}}}) \text{ by } G_{\rho_{\mathrm{SL,TL}}}(\psi) \stackrel{\mathrm{def}}{=} \\
 & \qquad \{s \in \Sigma_{\mathrm{SL}} \mid \forall t \in \Sigma_{\mathrm{TL}} : \ (s,t) \in \rho_{\mathrm{SL,TL}} : \ t \in \psi\}\} \\
 & \forall s \in \Sigma_{\mathrm{SL}} : \ s \in \phi : \ s \in G_{\rho_{\mathrm{SL,TL}}}(\psi) \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\} \\
 & \phi \subseteq G_{\rho_{\mathrm{SL,TL}}}(\psi)
\end{array}$$

and observe that $F_{\rho_{\mathrm{SL,TL}}}$ is universally disjunctive, cf. Sect. 3.2, because it has an upper adjoint, namely $G_{\rho_{\mathrm{SL,TL}}} \in (2^{\Sigma_{\mathrm{TL}}} \to 2^{\Sigma_{\mathrm{SL}}})$ defined by,[3] for all $\psi \subseteq \Sigma_{\mathrm{TL}}$,

$$G_{\rho_{\mathrm{SL,TL}}}(\psi) \stackrel{\mathrm{def}}{=} \{s \in \Sigma_{\mathrm{SL}} \mid \forall t \in \Sigma_{\mathrm{TL}} : \ (s,t) \in \rho_{\mathrm{SL,TL}} : \ t \in \psi\} \ . \tag{5.3}$$

It is this medium we will use for calculating with data representations in the world of predicate transformers and the observation above legitimates

Definition 5.1.1 (Data representation Galois connection). For given data representation relation $\rho_{\mathrm{SL,TL}} \subseteq \Sigma_{\mathrm{SL}} \times \Sigma_{\mathrm{TL}}$ the pair $(F_{\rho_{\mathrm{SL,TL}}}, G_{\rho_{\mathrm{SL,TL}}})$ as defined by (5.2) and (5.3) is called a *data representation Galois connection.*
□

It gets more complicated if a translation task is divided into several parts such that there is more than just one single data representation relating source and target states involved. For the moment let us assume given four state spaces, say, Σ_{SL}, Σ_{IL_1}, Σ_{IL_2} and Σ_{TL}, where IL is a shorthand for intermediate language, and consequently two data representation relations, say, $\rho_{\mathrm{SL,IL}_1} \subseteq \Sigma_{\mathrm{SL}} \times \Sigma_{\mathrm{IL}_1}$ and $\rho_{\mathrm{IL}_2,\mathrm{TL}} \subseteq \Sigma_{\mathrm{IL}_2} \times \Sigma_{\mathrm{TL}}$. We choose two different intermediate state spaces because the situation might emerge that the state space of the target language of a first translation step does not completely coincide with the state space of the source language of a second step. For a further discussion concerning this kind of inhomogeneity the reader is temporarily referred to Sect. 5.3. Each of the two relations, however, induces a data representation Galois connection, $(F_{\rho_{\mathrm{SL,IL}_1}}, G_{\rho_{\mathrm{SL,IL}_1}})$ and

[3] Note that $G_{\rho_{\mathrm{SL,TL}}}$ is *not* the left-image of $\rho_{\mathrm{SL,TL}}$.

$(F_{\rho_{\mathrm{IL}_2,\mathrm{TL}}}, G_{\rho_{\mathrm{IL}_2,\mathrm{TL}}})$. Formally, one can neither compose the relations nor the image functions because, as yet, the co-domain of the first has nothing in common with the domain of the second, though one would reasonably expect some commonness. As a remedy we embed both intermediate state spaces Σ_{IL_1} and Σ_{IL_2} into their union $\Sigma_{\mathrm{IL}} \stackrel{\mathrm{def}}{=} \Sigma_{\mathrm{IL}_1} \cup \Sigma_{\mathrm{IL}_2}$ such that by convention $\rho_{\mathrm{SL,IL}} = \rho_{\mathrm{SL,IL}_1} \subseteq \Sigma_{\mathrm{SL}} \times \Sigma_{\mathrm{IL}}$ and $\rho_{\mathrm{IL,TL}} = \rho_{\mathrm{IL}_2,\mathrm{TL}} \subseteq \Sigma_{\mathrm{IL}_2} \times \Sigma_{\mathrm{TL}}$. This embedding allows a sequential composition of relations in the classic way even if Σ_{IL_1} and Σ_{IL_2} are essentially different; Sect. 5.3 justifies this step. The composition $\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}$ is a subset of $\Sigma_{\mathrm{SL}} \times \Sigma_{\mathrm{TL}}$ and it is interesting to study how the composition of $F_{\rho_{\mathrm{SL,IL}}}$ and $F_{\rho_{\mathrm{IL,TL}}}$ gets along with this. For $\phi \subseteq \Sigma_{\mathrm{SL}}$ and $t \in \Sigma_{\mathrm{TL}}$ we calculate

$$
\begin{array}{ll}
 & t \in F_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}(\phi) \\
\Longleftrightarrow & \quad \{\text{Defintion (5.2)}\} \\
 & \exists s \in \Sigma_{\mathrm{SL}} :: \ (s,t) \in \rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}} \wedge s \in \phi \\
\Longleftrightarrow & \quad \{\text{Definition of relational composition}\} \\
 & \exists s \in \Sigma_{\mathrm{SL}} :: (\exists i \in \Sigma_{\mathrm{IL}} :: \ (s,i) \in \rho_{\mathrm{SL,IL}} \wedge (i,t) \in \rho_{\mathrm{IL,TL}}) \wedge s \in \phi \\
\Longleftrightarrow & \quad \{\text{Unnesting}\} \\
 & \exists s \in \Sigma_{\mathrm{SL}}, i \in \Sigma_{\mathrm{IL}} :: \ (s,i) \in \rho_{\mathrm{SL,IL}} \wedge (i,t) \in \rho_{\mathrm{IL,TL}} \wedge s \in \phi \\
\Longleftrightarrow & \quad \{\text{Nesting in other ways}\} \\
 & \exists i \in \Sigma_{\mathrm{IL}} :: \ (i,t) \in \rho_{\mathrm{IL,TL}} \wedge (\exists s \in \Sigma_{\mathrm{SL}} :: \ (s,i) \in \rho_{\mathrm{SL,IL}} \wedge s \in \phi) \\
\Longleftrightarrow & \quad \{\text{Definition (5.2)}\} \\
 & \exists i \in \Sigma_{\mathrm{IL}} :: \ (i,t) \in \rho_{\mathrm{IL,TL}} \wedge i \in F_{\rho_{\mathrm{SL,IL}}}(\phi) \\
\Longleftrightarrow & \quad \{\text{Definition (5.2) again}\} \\
 & t \in F_{\rho_{\mathrm{IL,TL}}}(F_{\rho_{\mathrm{SL,IL}}}(\phi)) \\
\Longleftrightarrow & \quad \{\text{Composition of predicate transformers}\} \\
 & t \in (F_{\rho_{\mathrm{IL,TL}}} \ ; F_{\rho_{\mathrm{SL,IL}}})(\phi) \ .
\end{array}
$$

Thus, the composition of the (embedded) data representation relations harmonizes with the composition of the corresponding image functions well and the above calculation proves

Lemma 5.1.1 (Composing the lower adjoints).

$$F_{\rho_{\mathrm{IL,TL}}} \ ; F_{\rho_{\mathrm{SL,IL}}} \ = \ F_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}$$

□

Since the functions F_ρ and G_ρ as specified by (5.2) and (5.3) form a Galois connection for every relation ρ of appropriate type the right-image of the compound, i.e. $F_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}$, has $G_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}$ as upper adjoint. Furthermore,

$$
\begin{array}{ll}
 & F_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}(\phi) \subseteq \psi \\
\Longleftrightarrow & \quad \{\text{Lemma 5.1.1 above}\}
\end{array}
$$

$$
\begin{array}{ll}
 & F_{\rho_{\mathrm{IL,TL}}}(F_{\rho_{\mathrm{SL,IL}}}(\phi)) \subseteq \psi \\
\Longleftrightarrow & \quad \{\text{Shunt twice}\} \\
 & \phi \subseteq G_{\rho_{\mathrm{SL,IL}}}(G_{\rho_{\mathrm{IL,TL}}}(\psi)) \\
\Longleftrightarrow & \quad \{\text{Composition of predicate transformers}\} \\
 & \phi \subseteq (G_{\rho_{\mathrm{SL,IL}}} \,;\, G_{\rho_{\mathrm{IL,TL}}})(\psi)
\end{array}
$$

such that we get the below result for free because upper adjoints are unique.

Lemma 5.1.2 (Composing the upper adjoints).

$$G_{\rho_{\mathrm{SL,IL}}} \,;\, G_{\rho_{\mathrm{IL,TL}}} \;=\; G_{\rho_{\mathrm{SL,IL}} \circ \rho_{\mathrm{IL,TL}}}$$

□

Note that the embedding of the data representation relations into the richer space, i.e. the union of the intermediate state spaces, allows the upper and lower adjoints to distribute over sequential composition of relations. Why this embedding is a sort of natural will be explained in Sect. 5.3.

5.2 Preserving Relative Correctness

Two programs, say π_1 and π_2, working on different state spaces induce two families of wrp-transformers and, ad hoc, one can neither relate the semantics of the programs nor reason about correctness preservation properties of their vertical or horizontal composition, because one is not even allowed to write something like "$\mathsf{wrp}_A.\pi_1 \leq \mathsf{wrp}_A.\pi_2$". Data representation Galois connections, as introduced before, are an adequate and convenient means to transfer predicates belonging to the state spaces of π_1 to predicates belonging to the state spaces of π_2 and vice versa. Hence, we will formulate the notion of "preservation of relative correctness" in this more complicated setting of inhomogeneous state spaces with the aid of such.

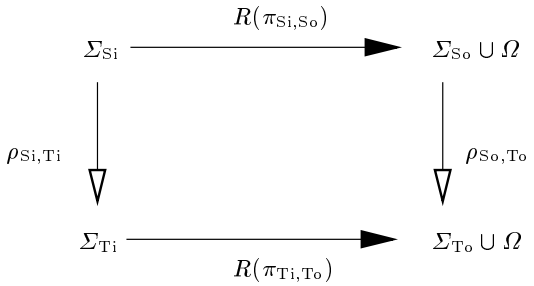

Fig. 5.1. Source and target program running on inhomogeneous state spaces.

For the actual discussion assume given two programs

$$R(\pi_{\mathrm{Si,So}}) \subseteq \Sigma_{\mathrm{Si}} \times (\Sigma_{\mathrm{So}} \cup \Omega) \quad \text{and} \quad R(\pi_{\mathrm{Ti,To}}) \subseteq \Sigma_{\mathrm{Ti}} \times (\Sigma_{\mathrm{To}} \cup \Omega) \;,$$

where, e.g., Si and To stand for "source language input" and "target language output" respectively. In order to be able to reason about the relationship between both programs we furthermore assume given two data representation relations

$$\rho_{\mathrm{Si,Ti}} \subseteq \Sigma_{\mathrm{Si}} \times \Sigma_{\mathrm{Ti}} \quad \text{and} \quad \rho_{\mathrm{So,To}} \subseteq \Sigma_{\mathrm{So}} \times \Sigma_{\mathrm{To}}$$

relating input and output states of the source and target language respectively, see Fig. 5.1 where the filled arrowheads represent relational semantics in the sense of (2.1) and the non-filled ones data representation relations in the sense of (5.1). Those relations induce the two data representation Galois connections

$$(F_{\rho_{\mathrm{Si,Ti}}}, G_{\rho_{\mathrm{Si,Ti}}}) \quad \text{and} \quad (F_{\rho_{\mathrm{So,To}}}, G_{\rho_{\mathrm{So,To}}})$$

as described before. Execution of each of the programs transforms states belonging to its own source language to states belonging to its own target language (or to erroneous outcomes) such that the basic notion of relative correctness changes slightly. Let us recall the definition for this inhomogeneous case. For predicates $\phi \subseteq \Sigma_{\mathrm{Si}}$ and predicates $\psi \subseteq \Sigma_{\mathrm{So}}$ and a set $A \subseteq \Omega$ of erroneous outcomes to be accepted, program $\pi_{\mathrm{Si,So}}$, for instance, is said to be *relatively correct w.r.t. ϕ, ψ and A*, still denoted by $\langle\phi\rangle\pi_{\mathrm{Si,So}}\langle\psi\rangle_A$ for short, iff

$$\forall s \in \Sigma_{\mathrm{Si}}, \sigma \in \Sigma_{\mathrm{So}} \cup \Omega :\ s \in \phi \wedge (s,\sigma) \in R(\pi_{\mathrm{Si,So}}) :\ \sigma \in \psi \cup A \ .$$

Consequently, the definition of the wrp-transformers changes also. In this case, $\mathsf{wrp}_A.\pi_{\mathrm{Si,So}}$ is of type $2^{\Sigma_{\mathrm{So}}} \to 2^{\Sigma_{\mathrm{Si}}}$ and is, for predicates $\psi \subseteq \Sigma_{\mathrm{So}}$, defined by

$$\mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \stackrel{\mathrm{def}}{=} \{s \in \Sigma_{\mathrm{Si}} \mid \forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega :\ (s,\sigma) \in R(\pi_{\mathrm{Si,So}}) :\ \sigma \in \psi \cup A\} \ .$$

Of course, $\mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi$ is still the weakest predicate ϕ such that $\langle\phi\rangle\pi_{\mathrm{Si,So}}\langle\psi\rangle_A$ holds. This observation is specified below and in particular it proves Lemma 4.3.1.

Lemma 5.2.1 ($\mathsf{wrp}.\pi$ *is* the weakest transformer, general case).

$$\phi \subseteq \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \quad \Longleftrightarrow \quad \langle\phi\rangle\pi_{\mathrm{Si,So}}\langle\psi\rangle_A$$

Proof.

$$\begin{array}{ll} & \langle\phi\rangle\pi_{\mathrm{Si,So}}\langle\psi\rangle_A \\ \Longleftrightarrow & \quad \{\text{Definition of relative correctness}\} \\ & \forall s \in \Sigma_{\mathrm{Si}}, \sigma \in \Sigma_{\mathrm{So}} \cup \Omega :\ s \in \phi \wedge (s,\sigma) \in R(\pi_{\mathrm{Si,So}}) :\ \sigma \in \psi \cup A \\ \Longleftrightarrow & \quad \{\text{Unnesting}\} \\ & \forall s \in \Sigma_{\mathrm{Si}} :\ s \in \phi : \\ & \qquad (\forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega :\ (s,\sigma) \in R(\pi_{\mathrm{Si,So}}) :\ \sigma \in \psi \cup A) \\ \Longleftrightarrow & \quad \{\text{Definition of } \mathsf{wrp}_A\} \\ & \forall s \in \Sigma_{\mathrm{Si}} :\ s \in \phi :\ s \in \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \\ \Longleftrightarrow & \quad \{\text{Set inclusion}\} \\ & \phi \subseteq \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \end{array}$$

□

Now, consider program $\pi_{\mathrm{Ti,To}}$ to be a translation of $\pi_{\mathrm{Si,So}}$. Preserving relative correctness means that all relative correctness assertions made for $\pi_{\mathrm{Si,So}}$ transfer to $\pi_{\mathrm{Ti,To}}$. Since these assertions are expressed in terms of predicates but, as yet, the states of the source and target language may have nothing in common, a direct formulation like in Theorem 4.3.1 does not work. One has to respect the representations of the predicates in question instead. We say that the translation of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ti,To}}$ *preserves relative correctness w.r.t.* A, $\rho_{\mathrm{Si,Ti}}$ *and* $\rho_{\mathrm{So,To}}$ or that $\pi_{\mathrm{Ti,To}}$ *implements* $\pi_{\mathrm{Si,So}}$ *w.r.t. preservation of relative correctness (PRC) w.r.t.* A, $\rho_{\mathrm{Si,Ti}}$ *and* $\rho_{\mathrm{So,To}}$ iff

$$\forall \phi \subseteq \Sigma_{\mathrm{Si}}, \psi \subseteq \Sigma_{\mathrm{So}} : \ \langle \phi \rangle \pi_{\mathrm{Si,So}} \langle \psi \rangle_A : \ \langle F_{\rho_{\mathrm{Si,Ti}}}(\phi) \rangle \pi_{\mathrm{Ti,To}} \langle F_{\rho_{\mathrm{So,To}}}(\psi) \rangle_A \ . \tag{5.4}$$

This formulation accords with the old: Each relative correctness assertion made for the source program by means of some predicates transfers to the target program by means of *representations* of the same predicates. (In particular, if all involved state spaces are equal, i.e. if each data representation relation is an identity, then the right-images degenerate to identities, too, such that there is a direct conformity.) As mentioned and discussed at length in Chap. 4, a refinement characterization of this notion, analogously to the one presented in Theorem 4.3.1, is very desirable. So let us look what we can extract from (5.4). Letting ϕ and ψ range over $2^{\Sigma_{\mathrm{Si}}}$ and $2^{\Sigma_{\mathrm{So}}}$ respectively, we calculate

$$\begin{array}{ll}
 & \forall \phi, \psi : \ \langle \phi \rangle \pi_{\mathrm{Si,So}} \langle \psi \rangle_A : \ \langle F_{\rho_{\mathrm{Si,Ti}}}(\phi) \rangle \pi_{\mathrm{Ti,To}} \langle F_{\rho_{\mathrm{So,To}}}(\psi) \rangle_A \\
\Longleftrightarrow & \quad \{\text{Alternative characterization, i.e. Lemma 5.2.1}\} \\
 & \forall \phi, \psi : \ \phi \subseteq \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi : \ F_{\rho_{\mathrm{Si,Ti}}}(\phi) \subseteq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}.(F_{\rho_{\mathrm{So,To}}}(\psi)) \\
\Longleftrightarrow & \quad \{(F_{\rho_{\mathrm{Si,Ti}}}, G_{\rho_{\mathrm{Si,Ti}}}) \text{ is a Galois connection}\} \\
 & \forall \phi, \psi : \ \phi \subseteq \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi : \ \phi \subseteq G_{\rho_{\mathrm{Si,Ti}}}(\mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}.(F_{\rho_{\mathrm{So,To}}}(\psi))) \\
\Longleftrightarrow & \quad \{\text{Indirect inequality}\} \\
 & \forall \psi :: \ \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \subseteq G_{\rho_{\mathrm{Si,Ti}}}(\mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}.(F_{\rho_{\mathrm{So,To}}}(\psi))) \\
\Longleftrightarrow & \quad \{\text{Composition of predicate transformers}\} \\
 & \forall \psi :: \ \mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi \subseteq (G_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \ ; F_{\rho_{\mathrm{So,To}}})(\psi) \\
\Longleftrightarrow & \quad \{\text{Lifted order on } \mathit{PTrans}\} \\
 & \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq G_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \ ; F_{\rho_{\mathrm{So,To}}} \ .
\end{array}$$

Hence, the known refinement characterization remains valid, the state spaces in question just have to be readjusted by means of data refinement Galois connections in order to be allowed to use the '$\leq$'-sign at all. Since both $(F_{\rho_{\mathrm{Si,Ti}}}, G_{\rho_{\mathrm{Si,Ti}}})$ and $(F_{\rho_{\mathrm{So,To}}}, G_{\rho_{\mathrm{So,To}}})$ are Galois connections there are several equivalent versions of this result which are collected in the theorem below. In particular it proves the first half of Theorem 4.3.1 (for the second half see Theorem 5.4.1).

Theorem 5.2.1 (Preserving relative correctness, refinement view). The translation of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ti,To}}$ preserves relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ti}}$ and $\rho_{\mathrm{So,To}}$ iff one of the following equivalent conditions holds.

1. $\mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq G_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}}$
2. $F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}}$
3. $F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \,;\, G_{\rho_{\mathrm{So,To}}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}$
4. $\mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \,;\, G_{\rho_{\mathrm{So,To}}} \leq G_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}$

Proof. The equivalence of 1 and 2 is obvious – just shunt $G_{\rho_{\mathrm{Si,Ti}}}$ to the left – and so is the equivalence of 3 and 4. We show the equivalence of 2 and 3 by the following ping-pong argument:[4]

$$
\begin{array}{ll}
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}} \\
\Longrightarrow & \quad \{\text{Monotonicity}\} \\
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \,;\, G_{\rho_{\mathrm{So,To}}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}} \,;\, G_{\rho_{\mathrm{So,To}}} \\
\Longrightarrow & \quad \{F_{\rho_{\mathrm{So,To}}} \,;\, G_{\rho_{\mathrm{So,To}}} \leq Id, \text{ see (3.4)}\} \\
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \,;\, G_{\rho_{\mathrm{So,To}}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \\
\Longrightarrow & \quad \{\text{Monotonicity}\} \\
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \,;\, G_{\rho_{\mathrm{So,To}}} \,;\, F_{\rho_{\mathrm{So,To}}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}} \\
\Longrightarrow & \quad \{Id \leq G_{\rho_{\mathrm{So,To}}} \,;\, F_{\rho_{\mathrm{So,To}}}, \text{ see (3.4)}\} \\
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}} \quad .
\end{array}
$$

Finally, the calculation before shows that 1 holds iff the translation of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ti,To}}$ preserves relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ti}}$ and $\rho_{\mathrm{So,To}}$.
□

5.3 Vertical and Horizontal Composition

The situation gets more interesting and closer to reality if compilers are combined vertically or horizontally (i.e. sequentially). In this section we show that a composed compiler inherits the correctness preservation properties of its components under certain and rather natural conditions.

5.3.1 Vertical composition

We first of all add two further programs to our setting, i.e. we assume given four programs, $\pi_{\mathrm{Si,So}}$, $\pi_{\mathrm{Ii,Io}}$, $\pi'_{\mathrm{Ii,Io}}$ and $\pi_{\mathrm{Ti,To}}$, the domains of which should be clear from their indices (again, 'Ii' resp. 'Io' is a shorthand for the input resp. output state space of an intermediate language). We furthermore assume that the input and output domains of the programs are coupled by the data representation relations $\rho_{\mathrm{Si,Ii}}$, $\rho_{\mathrm{So,Io}}$, $\rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{Io,To}}$ as shown in Fig. 5.2. Finally, imagine that we have two compilers at hand. The first one (correctly) translates programs $\pi_{\mathrm{Si,So}}$ of the

[4] Just a remark: In [27] D. Gries complains about scarce hints in calculational proofs. In particular the use of monotonicity should be stated more prominently in order to avoid confusion and to increase understanding. To adopt his proposal we will be rather careful to cite all needed properties but we will not hide simple and obvious steps behind a bulk of words. The second and the last step in the calculation also depend on monotonicity but we think that this is such obvious that one would be in search of monotonicity if it was mentioned.

actual source language to programs $\pi_{\mathrm{Ii,Io}}$ of a first intermediate language and a second one (correctly) translates programs $\pi'_{\mathrm{Ii,Io}}$ of a second intermediate language to programs $\pi_{\mathrm{Ti,To}}$ of the actual target language. We like to compose both compilers vertically in such ways that the compound (correctly) translates programs $\pi_{\mathrm{Si,So}}$ of the actual source language to programs $\pi_{\mathrm{Ti,To}}$ of the actual target language. To do so, however, the second compiler must be fed with outputs generated by the first and consequently the intermediate programs $\pi_{\mathrm{Ii,Io}}$ and $\pi'_{\mathrm{Ii,Io}}$ must be reasonably related in two respects. First of all there is the syntactical requirement that $\pi_{\mathrm{Ii,Io}}$ is a program of a sublanguage of $\pi'_{\mathrm{Ii,Io}}$'s language because otherwise the second compiler cannot cope with its input. Beside this, there are semantical requirements because even if the first compiler generates syntactically fitting programs the second compiler might have a different understanding of program semantics than the first. A wicked situation like this might arise in reality if parts of a compiler are developed independently by different groups, each of which has some further constraints concerning their language, some more knowledge concerning their second (source resp. target) language and also concerning their actual translation. However, typically $\pi_{\mathrm{Ii,Io}}$ should behave like $\pi'_{\mathrm{Ii,Io}}$ for certain inputs and the question to be discussed here is which requirements suffice to let the vertically composed compiler become correct.[5] As mentioned before the domains and codomains of $\pi_{\mathrm{Ii,Io}}$ and $\pi'_{\mathrm{Ii,Io}}$ need not to be the very same but it is clear that they must be reasonably related in order to let the composition work in the syntactical sense. Hence, without loss of generality, we keept them for equal because we can always take the union of the respective intermediate state spaces anyhow.

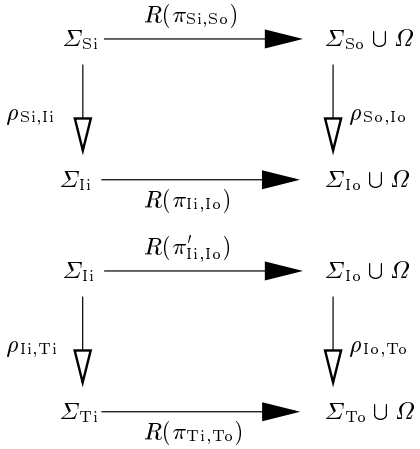

Fig. 5.2. The even more inhomogeneous scenario: Vertical composition of correct translations.

Let us start a naive proof which hopefully brings to light the searched requirements. Having a look at Theorem 5.2.1, e.g. version 2, the goal amounts to showing that

$$F_{\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \; ; F_{\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}} \quad , \tag{5.5}$$

[5] To summarize, we distinguish between $\pi_{\mathrm{Ii,Io}}$ and $\pi'_{\mathrm{Ii,Io}}$ to show that they do not need to be equal, they only have to be sensibly related and the task is to find these requirements.

saying that the translation of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ti,To}}$ preserves relative correctness w.r.t. A and the composition of the involved data representation relations. A point-wise proof could be performed by unrolling the definitions. Though quite accurate this proceeding would barely be clean and comprehensible (in the sense of readability); furthermore the abstractions made so far would be rather useless. Instead, for the left hand side we may calculate

$$\begin{array}{ll} & F_{\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \\ = & \quad \{\text{Lemma 5.1.1}\} \\ & F_{\rho_{\mathrm{Ii,Ti}}} \; ; F_{\rho_{\mathrm{Si,Ii}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \\ \leq & \quad \{\pi_{\mathrm{Ii,Io}} \text{ implements } \pi_{\mathrm{Si,So}}, \text{ monotonicity}\} \\ & F_{\rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \; ; F_{\rho_{\mathrm{So,Io}}} \quad , \end{array}$$

and similarly for the right hand side

$$\begin{array}{ll} & \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \; ; F_{\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}} \\ = & \quad \{\text{Lemma 5.1.1}\} \\ & \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \; ; F_{\rho_{\mathrm{Io,To}}} \; ; F_{\rho_{\mathrm{So,Io}}} \\ \geq & \quad \{\pi_{\mathrm{Ti,To}} \text{ implements } \pi'_{\mathrm{Ii,Io}}, \text{ monotonicity}\} \\ & F_{\rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \; ; F_{\rho_{\mathrm{So,Io}}} \end{array}$$

where the monotonicity of the composition operator is exploited twice. Hence, we were done if we could establish

$$F_{\rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \; ; F_{\rho_{\mathrm{So,Io}}} \leq F_{\rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \; ; F_{\rho_{\mathrm{So,Io}}} \quad . \tag{5.6}$$

(Note that, in (5.6), the data representation relations $\rho_{\mathrm{Si,Ii}}$ and $\rho_{\mathrm{Io,To}}$ have vanished.) Now, a pointwise proof could be started from here but we could also continue by exploiting monotonicity twice again: Inequality (5.6) obviously follows if we had

$$\mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \leq \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \quad . \tag{5.7}$$

But this is a far too strong requirement though quite satisfactory from our abstract view where we try to calculate as abstractly as possible. In particular, inequality (5.7) requires $\pi_{\mathrm{Ii,Io}}$ and $\pi'_{\mathrm{Ii,Io}}$ to behave "equal" for every initial state. All involved data representation relations have vanished now but it turns out that one can restrict attention to a certain class of initial states. Anyway, this algebraic calculation establishes

Lemma 5.3.1 (Vertical composition). If the translations of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ii,Io}}$ and of $\pi'_{\mathrm{Ii,Io}}$ to $\pi_{\mathrm{Ti,To}}$ preserve relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}}$ and $\rho_{\mathrm{So,Io}}$ resp. w.r.t. A, $\rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{Io,To}}$ then $\pi_{\mathrm{Ti,To}}$ implements $\pi_{\mathrm{Si,So}}$ in the sense of preservation of relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}$ if

$$\mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \leq \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \quad .$$

□

In order to find weaker requirements one could prove (5.6) directly by unrolling the definitions. We start to expand the left hand side: For all $t \in \Sigma_{\mathrm{Ti}}$ and $\psi \in 2^{\Sigma_{\mathrm{So}}}$:

$$
\begin{array}{ll}
 & t \in F_{\rho_{\mathrm{Ii,Ti}}}(\mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}}.(F_{\rho_{\mathrm{So,Io}}}(\psi))) : \\
\iff & \quad \{\text{Definition of } F_{\rho_{\mathrm{Ii,Ti}}}\} \\
 & \exists s \in \Sigma_{\mathrm{Ii}} :: \ (s,t) \in \rho_{\mathrm{Ii,Ti}} \wedge s \in \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}}.(F_{\rho_{\mathrm{So,Io}}}(\psi)) \\
\iff & \quad \{\text{Definition of } \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}}\} \\
 & \exists s \in \Sigma_{\mathrm{Ii}} :: \ (s,t) \in \rho_{\mathrm{Ii,Ti}} \wedge \\
 & \qquad\qquad \forall \sigma \in \Sigma_{\mathrm{Io}} \cup \Omega : \ (s,\sigma) \in R(\pi_{\mathrm{Ii,Io}}) : \ \sigma \in F_{\rho_{\mathrm{So,Io}}}(\psi) \cup A \ .
\end{array}
$$

Similarly, for the right hand side one obtains

$$
\begin{array}{ll}
 & t \in F_{\rho_{\mathrm{Ii,Ti}}}(\mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}}.(F_{\rho_{\mathrm{So,Io}}}(\psi))) : \\
\iff & \quad \{\text{Analogous steps}\} \\
 & \exists s \in \Sigma_{\mathrm{Ii}} :: \ (s,t) \in \rho_{\mathrm{Ii,Ti}} \\
 & \qquad\qquad \forall \sigma \in \Sigma_{\mathrm{Io}} \cup \Omega : \ (s,\sigma) \in R(\pi'_{\mathrm{Ii,Io}}) : \ \sigma \in F_{\rho_{\mathrm{So,Io}}}(\psi) \cup A \ .
\end{array}
$$

A sufficient condition asserting that the upper implies the lower is that

$$
R(\pi'_{\mathrm{Ii,Io}})|_{\rho^{-1}_{\mathrm{Ii,Ti}}(\Sigma_{\mathrm{Ti}})} \subseteq R(\pi_{\mathrm{Ii,Io}}) \ , \tag{5.8}
$$

where for a relation $R \subseteq A \times B$ and $S \subseteq A$ the *restriction of R to S* is defined by

$$
R|_S \stackrel{\text{def}}{=} \{(a,b) \mid a \in S \wedge (a,b) \in R\} \ .
$$

Let us have a closer look at (5.8). Each computation of $\pi'_{\mathrm{Ii,Io}}$ starting in a state $s \in \Sigma_{\mathrm{Ii}}$ which has a representation $t \in \Sigma_{\mathrm{Ti}}$ must be a possible computation of $\pi_{\mathrm{Ii,Io}}$. This restriction is natural in the following sense. If one considers the translation of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ti,To}}$, which is the final goal, one is interested in states $s \in \Sigma_{\mathrm{Si}}$ and $s' \in \Sigma_{\mathrm{Ti}}$ which are related by $\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$, i.e. such that $(s,s') \in \rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$. But this implies the existence of an intermediate state $i \in \rho^{-1}_{\mathrm{Ii,Ti}}(\Sigma_{\mathrm{Ti}})$ so we only have to care about those. (This also implies the existence of an intermediate state $i \in \rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}})$ but we cannot conclude this from here because $\rho_{\mathrm{Si,Ii}}$ has vanished in (5.6) due to the exploitation of monotonicity.)

What we have learned so far is that (5.8) implies (5.6) and hence (5.5). But we could have started our considerations concerning vertical composition also with version 1, 3 or 4 of Theorem 5.2.1 instead of 2. And indeed, after some very similar calculations, we would have obtained other sufficient conditions. Analogously to the first abstraction step (5.6) they read

1. $G_{\rho_{\mathrm{Si,Ii}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \,;\, F_{\rho_{\mathrm{So,Io}}} \leq G_{\rho_{\mathrm{Si,Ii}}} \,;\, \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \,;\, F_{\rho_{\mathrm{So,Io}}}$,
3. $F_{\rho_{\mathrm{Ii,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \,;\, G_{\rho_{\mathrm{Io,To}}} \leq F_{\rho_{\mathrm{Ii,Ti}}} \,;\, \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \,;\, G_{\rho_{\mathrm{Io,To}}}$, and
4. $G_{\rho_{\mathrm{Si,Ii}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Ii,Io}} \,;\, G_{\rho_{\mathrm{Io,To}}} \leq G_{\rho_{\mathrm{Si,Ii}}} \,;\, \mathsf{wrp}_A.\pi'_{\mathrm{Ii,Io}} \,;\, G_{\rho_{\mathrm{Io,To}}}$.

Each of the inequalities above implies preservation of relative correctness formulated in its corresponding version of Theorem 5.2.1. Again, from here one could start a pointwise proof yielding requirements in terms of the relational semantics as done for 2 before. They are given in the following lemma where we keep the

numbering from Theorem 5.2.1 to show where they arise from (thus the requirements appear twice). A proof is omitted because it gives no further insight and is completely similar to the one just given.

Lemma 5.3.2 (Vertical composition, strong requirements). If the translations of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ii,Io}}$ and of $\pi'_{\mathrm{Ii,Io}}$ to $\pi_{\mathrm{Ti,To}}$ preserve relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}}$ and $\rho_{\mathrm{So,Io}}$ resp. w.r.t. A, $\rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{Io,To}}$ then $\pi_{\mathrm{Ti,To}}$ implements $\pi_{\mathrm{Si,So}}$ in the sense of preservation of relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}$ if one of the following conditions holds.

1. $R(\pi'_{\mathrm{Ii,Io}})|_{\rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}})} \subseteq R(\pi_{\mathrm{Ii,Io}})$.
2. $R(\pi'_{\mathrm{Ii,Io}})|_{\rho^{-1}_{\mathrm{Ii,Ti}}(\Sigma_{\mathrm{Ti}})} \subseteq R(\pi_{\mathrm{Ii,Io}})$.
3. $R(\pi'_{\mathrm{Ii,Io}})|_{\rho^{-1}_{\mathrm{Ii,Ti}}(\Sigma_{\mathrm{Ti}})} \subseteq R(\pi_{\mathrm{Ii,Io}})$.
4. $R(\pi'_{\mathrm{Ii,Io}})|_{\rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}})} \subseteq R(\pi_{\mathrm{Ii,Io}})$.

□

Lemma 5.3.2 presents the "weakest" demands that naturally arise in a pointwise proof after two proximate exploitations of monotonicity. We preferred the proceeding presented here in order to stress and demonstrate that not only predicate transformers provide abstractions of relational semantics, this is obvious, but also the used framework in some sense. Pointwise reasoning without, say, typing constraints leaves much more calculational freedom and in each typical algebraic step potential useful information might get lost; Sect. 5.5 is concerned with observations like these.

However, as mentioned before, in the very end we are only interested in states $s \in \Sigma_{\mathrm{Si}}$ and $s' \in \Sigma_{\mathrm{Ti}}$ which are related by $\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$. In particular s and s' are states which have resp. are representations in Σ_{Ii}. Thus, it should be possible to restrict attention to $\rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}}) \cap \rho^{-1}_{\mathrm{Ii,Ti}}(\Sigma_{\mathrm{Ti}})$. To be even more precise, not the entire state space Σ_{Ti} is of interest but only the domain of $R(\pi_{\mathrm{Ti,To}})$; remember that we did not demand the relational semantics to be total. Indeed, this is also provable but, unfortunately and obviously, not in a purely algebraic fashion because the use of monotonicity always lets some of the involved data representation relations or relational semantics vanish.

Theorem 5.3.1 (Vertical composition, weak requirements). If the translations of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ii,Io}}$ and of $\pi'_{\mathrm{Ii,Io}}$ to $\pi_{\mathrm{Ti,To}}$ preserve relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}}$ and $\rho_{\mathrm{So,Io}}$ resp. w.r.t. A, $\rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{Io,To}}$ then $\pi_{\mathrm{Ti,To}}$ implements $\pi_{\mathrm{Si,So}}$ in the sense of preservation of relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}$ and $\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}$ if

$$R(\pi'_{\mathrm{Ii,Io}}) \cap (R_{\mathrm{in}} \times (R_{\mathrm{out}} \cup \Omega \setminus A)) \;\subseteq\; R(\pi_{\mathrm{Ii,Io}}) \;\;,$$

where

$$R_{\mathrm{in}} \;=\; \rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}}) \cap \rho^{-1}_{\mathrm{Ii,Ti}}(R(\pi_{\mathrm{Ti,To}})^{-1}(\Sigma_{\mathrm{To}} \cup \Omega))$$

and

$$R_{\mathrm{out}} \;=\; \rho^{-1}_{\mathrm{Io,To}}(\Sigma_{\mathrm{To}}) \;\;.$$

The *Proof* is omitted here because Sect. 5.4 is concerned with the relational treatment and it seems appropriate to present the proof there.
□

We like to remark that each of the requirements 1 to 4 in Lemma 5.3.2 implies the requirement made in Theorem 5.3.1. Thus, the slightly more abstract calculation lets us obtain requirements – in a more or less natural manner – which are unfortunately stronger than sufficient.

5.3.2 Horizontal composition

The situation is essentially different now: Not implementations are to be combined but programs. Generally speaking, the question studied in this section is the following (for the sake of comprehension we modify the picture and the naming of the state spaces, see now Fig. 5.3). Suppose that $\pi_{\mathrm{Ti,T'}}$ and $\pi_{\mathrm{T',To}}$ implement $\pi_{\mathrm{Si,S'}}$ and $\pi_{\mathrm{S',So}}$ respectively. This time we take each regular result delivered by $\pi_{\mathrm{Si,S'}}$ resp. $\pi_{\mathrm{Ti,T'}}$ for an input to $\pi_{\mathrm{S',So}}$ resp. $\pi_{\mathrm{T',To}}$. The exercise is to find sufficient requirements assuring that the sequential composition of $\pi_{\mathrm{Ti,T'}}$ and $\pi_{\mathrm{T',To}}$ implements the sequential composition of $\pi_{\mathrm{Si,S'}}$ and $\pi_{\mathrm{S',So}}$. Again it seems that the intermediate state spaces must be equal but due to the known syntactical requirements that the second compiler should cope with its inputs we assume that they are insofar reasonalbly related that we can again take their union such that we can assume them to be equal anyhow. A mismatch like this pops up in similar situations as mentioned before: Two compilers are built seperately from each other and each group uses different data representation relations on slightly different states. Thus, the searched requirements that let the seqentially composed compiler become correct are made on the data representation relation level. Remember that, for vertical composition, similar demands were made on the intermediate programs.

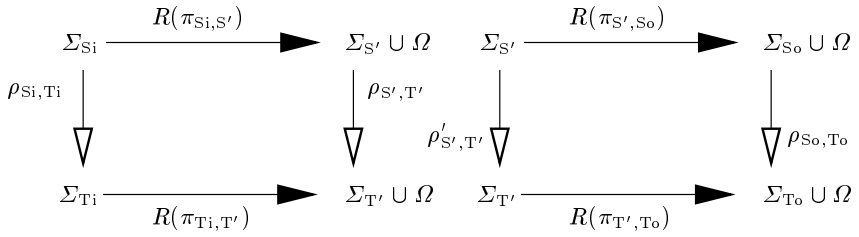

Fig. 5.3. Horizontal composition of correct translations.

In order to perform the task one has to introduce the sequential composition $\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}$ of programs $\pi_{\mathrm{Si,S'}}$ and $\pi_{\mathrm{S',So}}$ resp. its semantics $R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}})$ first.[6] Intuitively, each regular output generated by $\pi_{\mathrm{Si,S'}}$ serves as an input of $\pi_{\mathrm{S',So}}$ and each irregular outcome of $\pi_{\mathrm{Si,S'}}$ is of course the outcome of the compound because the computation either terminates irregularly or does not terminate at all such that the second program cannot even start; in this sense the composition is *error*

[6] The more common symbol ';' is already assigned.

strict. More formally, the meaning of the sequential composition, $R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}) \subseteq \Sigma_{\mathrm{Si}} \times (\Sigma_{\mathrm{So}} \cup \Omega)$, is defined as

$$\begin{aligned} & R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}) \ = \\ & \qquad \{(s,\sigma) \mid s \in \Sigma_{\mathrm{Si}} \ \wedge \sigma \in \Sigma_{\mathrm{So}} \cup \Omega \ \wedge \\ & \qquad\qquad \exists s' \in \Sigma_{\mathrm{S'}} :: \ (s,s') \in R(\pi_{\mathrm{Si,S'}}) \wedge (s',\sigma) \in R(\pi_{\mathrm{S',So}})\} \\ & \cup \ \{(s,\omega) \mid s \in \Sigma_{\mathrm{Si}} \wedge \omega \in \Omega \wedge (s,\omega) \in R(\pi_{\mathrm{Si,S'}})\} \ . \end{aligned}$$

As done for data representation relations let us first of all have a look how wrp-transformers get along with sequential composition. For $\psi \subseteq \Sigma_{\mathrm{So}}$ and $s \in \Sigma_{\mathrm{Si}}$ we calculate

$$\begin{aligned} & s \in \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}.\psi \\ \Longleftrightarrow & \quad \{\text{Definition of } \mathsf{wrp}_A\} \\ & \forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega : \ (s,\sigma) \in R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}) : \ \sigma \in \psi \cup A \\ \Longleftrightarrow & \quad \{\text{Definition of sequential composition} \\ & \qquad \text{and splitting the range}\} \\ & (\forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega : \ \sigma \in \Omega \wedge (s,\sigma) \in R(\pi_{\mathrm{Si,S'}}) : \ \sigma \in \psi \cup A) \ \wedge \\ & (\forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega : \\ & \quad \exists s' \in \Sigma_{\mathrm{S'}} :: \ (s,s') \in R(\pi_{\mathrm{Si,S'}}) \wedge (s',\sigma) \in R(\pi_{\mathrm{S',So}}) : \\ & \qquad \sigma \in \psi \cup A) \\ \Longleftrightarrow & \quad \{\exists\text{-introduction, unnesting and naming conventions}\} \\ & (\forall \omega \in \Omega : \ (s,\omega) \in R(\pi_{\mathrm{Si,S'}}) : \ \omega \in A) \ \wedge \\ & (\forall s' \in \Sigma_{\mathrm{S'}} : \ (s,s') \in R(\pi_{\mathrm{Si,S'}}) : \\ & \quad \forall \sigma \in \Sigma_{\mathrm{So}} \cup \Omega : \ (s',\sigma) \in R(\pi_{\mathrm{S',So}}) : \ \sigma \in \psi \cup A) \\ \Longleftrightarrow & \quad \{\text{Naming conventions, } \forall \text{ distributes over } \wedge \\ & \qquad \text{and definition of } \mathsf{wrp}_A\} \\ & \forall \sigma \in \Sigma_{\mathrm{S'}} \cup \Omega : \ (s,\sigma) \in R(\pi_{\mathrm{Si,S'}}) : \ \sigma \in \mathsf{wrp}_A.\pi_{\mathrm{S',So}}.\psi \cup A \\ \Longleftrightarrow & \quad \{\text{Definition of } \mathsf{wrp}_A \text{ again}\} \\ & s \in \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}}.(\mathsf{wrp}_A.\pi_{\mathrm{S',So}}.\psi) \end{aligned}$$

and observe that – as expected – the wrp_A-transformers distribute over sequential composition of programs. Vocalized: The weakest relative precondition of a sequential composition ist the weakest relative precondition of the first component which establishes the weakest relative precondition of the second.

Lemma 5.3.3 (Sequential Composition of programs).

$$\mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \ = \ \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}} \ .$$

□

Let us now become more specific what horizontal composition concerns and assume that $\pi_{\mathrm{Ti,T'}}$ resp. $\pi_{\mathrm{T',To}}$ implements $\pi_{\mathrm{Si,S'}}$ resp. $\pi_{\mathrm{S',So}}$ in the sense of preservation of relative correctness w.r.t. A and the respective data representation relations. Being in search for sufficient requirements implying that $\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}}$

implements $\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}$ w.r.t. A and the data representation relations $\rho_{\mathrm{Si,Ti}}$ and $\rho_{\mathrm{So,To}}$ we have a look at Theorem 5.2.1, e.g. version 2, and like to prove

$$F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}} \,;\, F_{\rho_{\mathrm{So,To}}} \quad . \tag{5.9}$$

Simultaneously we start our calculations from the left hand side,

$$\begin{array}{ll} & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}} \\ = & \quad \{\text{Lemma 5.3.3}\} \\ & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \\ \leq & \quad \{\pi_{\mathrm{Ti,T'}} \text{ implements } \pi_{\mathrm{Si,S'}}, \text{ monotonicity}\} \\ & \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \,;\, F_{\rho_{\mathrm{S',T'}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \quad , \end{array}$$

and from the right hand side,

$$\begin{array}{ll} & \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}} \,;\, F_{\rho_{\mathrm{So,To}}} \\ \geq & \quad \{\text{Lemma 5.3.3}\} \\ & \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{T',To}} \,;\, F_{\rho_{\mathrm{So,To}}} \\ \geq & \quad \{\pi_{\mathrm{T',To}} \text{ implements } \pi_{\mathrm{S',So}}, \text{ monotonicity}\} \\ & \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \,;\, F_{\rho'_{\mathrm{S',T'}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \quad , \end{array}$$

yielding

$$\mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \,;\, F_{\rho_{\mathrm{S',T'}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \,;\, F_{\rho'_{\mathrm{S',T'}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{S',So}} \tag{5.10}$$

as a sufficient condition. (Note that, again, some of the involved relations have vanished due to the exploitation of monotonicity.) Following the lines from above one could apply monotonicity twice yielding $F_{\rho_{\mathrm{S',T'}}} \leq F_{\rho'_{\mathrm{S',T'}}}$ as yet a stronger sufficient condition implying (5.10). Consequently this proves

Lemma 5.3.4 (Horizontal composition). If the two translations of $\pi_{\mathrm{Si,S'}}$ to $\pi_{\mathrm{Ti,T'}}$ and of $\pi_{\mathrm{S',So}}$ to $\pi_{\mathrm{T',To}}$ preserve relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ti}}$ and $\rho_{\mathrm{S',T'}}$ resp. $\rho'_{\mathrm{S',T'}}$ and $\rho_{\mathrm{So,To}}$, then $\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}}$ implements $\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}$ in the sense of preservation of relative correctness w.r.t. A, $\rho_{\mathrm{Si,Ti}}$ and $\rho_{\mathrm{So,To}}$ if

$$F_{\rho_{\mathrm{S',T'}}} \leq F_{\rho'_{\mathrm{S',T'}}}$$

□

But again this would be a very strong requirement because it again suffices to respect certain states in $\Sigma_{\mathrm{S'}}$. Instead we start a pointwise proof of (5.10). Since it is always the same old story we omit some intermediate steps:

$$\begin{array}{ll} & s \in \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}}.(F_{\rho_{\mathrm{S',T'}}}(\mathsf{wrp}_A.\pi_{\mathrm{S',So}}.\psi)) \\ \Longleftrightarrow & \quad \{\text{Carefully expand the definitions}\} \\ & \forall \sigma \in \Sigma_{\mathrm{T'}} \cup \Omega : \ (s,\sigma) \in R(\pi_{\mathrm{Ti,T'}}) : \\ & \qquad \exists t \in \Sigma_{\mathrm{S'}} :: \ ((t,\sigma) \in \rho_{\mathrm{S',T'}} \wedge t \in \mathsf{wrp}_A.\pi_{\mathrm{S',So}}.\psi) \end{array}$$

for the left hand side and

$$
\begin{array}{ll}
& s \in \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}}.(F_{\rho'_{\mathrm{S'},\mathrm{T'}}}(\mathsf{wrp}_A.\pi_{\mathrm{S'},\mathrm{So}}.\psi)) \\
\Longleftrightarrow & \quad \{\cdots\} \\
& \forall \sigma \in \Sigma_{\mathrm{T'}} \cup \Omega : \ (s,\sigma) \in R(\pi_{\mathrm{Ti,T'}}) : \\
& \qquad\qquad \exists t \in \Sigma_{\mathrm{S'}} :: \ ((t,\sigma) \in \rho'_{\mathrm{S'},\mathrm{T'}} \wedge t \in \mathsf{wrp}_A.\pi_{\mathrm{S'},\mathrm{So}}.\psi)
\end{array}
$$

for the right hand side. Seeking for sufficient conditions such that the upper implies the lower yields

$$\rho^{-1}_{\mathrm{S'},\mathrm{T'}}|_{R(\pi_{\mathrm{Ti,T'}})(\Sigma_{\mathrm{Ti}})\cap\Sigma_{\mathrm{T'}}} \subseteq \rho'^{-1}_{\mathrm{S'},\mathrm{T'}} \ . \tag{5.11}$$

The mentioned needs so far are barely astonishing because whenever, say t', is a possible regular result of $\pi_{\mathrm{Ti,T'}}$ which is a representation of, say $s' \in \Sigma_{\mathrm{S'}}$, we expect both, s' and t', to be possible inputs for $\pi_{\mathrm{S'},\mathrm{So}}$ and $\pi_{\mathrm{T'},\mathrm{To}}$ respectively which are also related by $\rho'_{\mathrm{S'},\mathrm{T'}}$. Otherwise one can hardly guarantee anything because, roughly speaking, the inputs of $\pi_{\mathrm{S'},\mathrm{So}}$ and $\pi_{\mathrm{T'},\mathrm{To}}$ have nothing in common from their point of view though they have from the view of $\pi_{\mathrm{Si,S'}}$ and $\pi_{\mathrm{Ti,T'}}$. Requirement (5.11), however, lets regular results obtained by $\pi_{\mathrm{S'},\mathrm{So}}$ resp. $\pi_{\mathrm{T'},\mathrm{To}}$ started in s' resp. t' be related also by $\rho_{\mathrm{So},\mathrm{To}}$ because $\pi_{\mathrm{T'},\mathrm{To}}$ implements $\pi_{\mathrm{S'},\mathrm{So}}$.

Again, we might have started our calculations with another equivalent version of preservation of relative correctness, cf. Theorem 5.2.1, and again we would have obtained several other sufficient conditions. Starting from versions 1, 3 resp. 4 the corresponding proximate candidates read as follows.

1. $$G_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \ ; F_{\rho_{\mathrm{S'},\mathrm{T'}}} \ ; G_{\rho'_{\mathrm{S'},\mathrm{T'}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{T'},\mathrm{To}} \ ; F_{\rho_{\mathrm{So},\mathrm{To}}}$$
$$\leq \ G_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Ti,T'}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{T'},\mathrm{To}} \ ; F_{\rho_{\mathrm{So},\mathrm{To}}}$$

3. $$F_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{S'},\mathrm{So}} \ ; G_{\rho_{\mathrm{So},\mathrm{To}}}$$
$$\leq \ F_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \ ; G_{\rho_{\mathrm{S'},\mathrm{T'}}} \ ; F_{\rho'_{\mathrm{S'},\mathrm{T'}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{S'},\mathrm{So}} \ ; G_{\rho_{\mathrm{So},\mathrm{To}}}$$

4. $$\mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \ ; G_{\rho'_{\mathrm{S'},\mathrm{T'}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{T'},\mathrm{To}}$$
$$\leq \ \mathsf{wrp}_A.\pi_{\mathrm{Si,S'}} \ ; G_{\rho_{\mathrm{S'},\mathrm{T'}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{T'},\mathrm{To}}$$

Version 4 looks somewhat similar to 2 above and so does its unrolled version. It takes a few steps to see that 4 follows if

$$\rho_{\mathrm{S'},\mathrm{T'}}|_{R(\pi_{\mathrm{Si,S'}})(\Sigma_{\mathrm{Si}})\cap\Sigma_{\mathrm{S'}}} \subseteq \rho'_{\mathrm{S'},\mathrm{T'}} \ , \tag{5.12}$$

and this requirement can be illustrated as follows. If $\pi_{\mathrm{Si,S'}}$ delivers a regular result, say t, then each regular result, say t', computed by $\pi_{\mathrm{Ti,T'}}$ is related to t by $\rho_{\mathrm{S'},\mathrm{T'}}$ because $\pi_{\mathrm{Ti,T'}}$ implements $\pi_{\mathrm{Si,S'}}$. By (5.12) t and t' are related also by $\rho'_{\mathrm{S'},\mathrm{T'}}$ such that each regular result computed by $\pi_{\mathrm{S'},\mathrm{So}}$, started in t, is related to each regular result computed by $\pi_{\mathrm{T'},\mathrm{To}}$, started in t', by $\rho_{\mathrm{So},\mathrm{To}}$ because $\pi_{\mathrm{T'},\mathrm{To}}$ implements $\pi_{\mathrm{S'},\mathrm{So}}$.

Natural candidates implying 1 resp. 3 are, for instance, $F_{\rho_{\mathrm{S'},\mathrm{T'}}} \ ; G_{\rho'_{\mathrm{S'},\mathrm{T'}}} \leq Id$ resp. $Id \leq G_{\rho_{\mathrm{S'},\mathrm{T'}}} \ ; F'_{\rho_{\mathrm{S'},\mathrm{T'}}}$ but this is again a too strong requirement (but of course sufficient). One could unroll the definitions and start a pointwise calculation but, complicated as they are, in this case a direct calculation for their counterparts in Theorem 5.2.1 seems preferable.

However, there are weaker demands on the relationship between the intermediate relations $\rho'_{\mathrm{S'},\mathrm{T'}}$ and $\rho_{\mathrm{S'},\mathrm{T'}}$ but, naturally and in some sense obviously, they

cannot be found in a more or less purely algebraic or at least natural fashion because the exploitation of monotonicity lets information vanish that might have been helpful in a later stage (and because these situations are really wicked). The following requirements look like a rabbit drawn from a magician's hat but they can be obtained in a natural way from a relational point of view and, luckily, (5.11) resp. (5.12) is stronger so that the more algebraic calculation at least has a similar flavor.

Theorem 5.3.2 (Horizontal composition, weak requirements). If the two translations of $\pi_{\mathrm{Si},\mathrm{S}'}$ to $\pi_{\mathrm{Ti},\mathrm{T}'}$ and of $\pi_{\mathrm{S}',\mathrm{So}}$ to $\pi_{\mathrm{T}',\mathrm{To}}$ preserve relative correctness w.r.t. A, $\rho_{\mathrm{Si},\mathrm{Ti}}$ and $\rho_{\mathrm{S}',\mathrm{T}'}$ resp. $\rho'_{\mathrm{S}',\mathrm{T}'}$ and $\rho_{\mathrm{So},\mathrm{To}}$, then $\pi_{\mathrm{Ti},\mathrm{T}'} \bullet \pi_{\mathrm{T}',\mathrm{To}}$ implements $\pi_{\mathrm{Si},\mathrm{S}'} \bullet \pi_{\mathrm{S}',\mathrm{So}}$ in the sense of preservation of relative correctness w.r.t. A, $\rho_{\mathrm{Si},\mathrm{Ti}}$ and $\rho_{\mathrm{So},\mathrm{To}}$ if

$$\rho_{\mathrm{S}',\mathrm{T}'} \cap (R_{\mathrm{in}} \times R_{\mathrm{out}}) \subseteq \rho'_{\mathrm{S}',\mathrm{T}'} \ ,$$

where

$$R_{\mathrm{in}} \stackrel{\mathrm{def}}{=} R(\pi_{\mathrm{Si},\mathrm{S}'})(\Sigma_{\mathrm{Si}})$$

and

$$R_{\mathrm{out}} \stackrel{\mathrm{def}}{=} R(\pi_{\mathrm{Ti},\mathrm{T}'})(\rho_{\mathrm{Si},\mathrm{Ti}}(\Sigma_{\mathrm{Si}})) \cap R(\pi_{\mathrm{T}',\mathrm{To}})^{-1}(\Sigma_{\mathrm{To}} \cup \Omega) \ .$$

Again, the *Proof* is shifted to Sect. 5.4.
□

The restricting set of states has even more decreased so let us briefly extend the illustrations made so far. One is not interested in the entire set of regular outcomes produced by $\pi_{\mathrm{Ti},\mathrm{T}'}$ but only in states which are outcomes of $\pi_{\mathrm{Ti},\mathrm{T}'}$-computations starting in initial states which are representations of some states of Σ_{Si}. Furthermore it suffices to respect states in which $\pi_{\mathrm{T}',\mathrm{To}}$ can start at all. Additionally, only regular results of $\pi_{\mathrm{Si},\mathrm{S}'}$ have to be considered for which all representations are possible initial states of $\pi_{\mathrm{T}',\mathrm{To}}$.

At the end of the predicate transformer discussion concerning inhomogeneity and compositionality we like to make the following remarks. What should have become clearer is that relative correctness is a "transitive notion" in the sense that – sensible correlations presumed – relative-correctness-preserving translations can be composed vertically and sequentially, and this even in the presence of different state spaces. For PPC and PTC this shows already in the relational semantics, see Figs. 4.1 and 4.2, but for preservation of relative correctness in terms of a relational semantics, see Fig. 4.3, this is less obvious. As mentioned there, no inclusion on any level can be easily recognized at first glance. Separating regular states from irregular outcomes, some of which to be accepted and others not, and reasoning in terms of wrp-transformers and data representation Galois connections instead of relations, however, unveils that PRC is a transitive notion, transitive w.r.t. '$\leq$'. This insight is indeed an achievement because it formally and in a way beautifully captures the wicked situations that come across in the real world.

5.4 The Relational Setting

Describing the semantics of programs by means of relations has a lot of advantages. Firstly a relational semantics is very close to the very basic operational behavior and secondly it is also close to the intuition of the average programmer so errors in the formal semantics resp. errors due to misunderstandings, hopefully, become rare. Furthermore relations are such fundamental that nearly everyone involved in programming should be able to reason about program properties in terms of such.

But, as we have seen, a relational semantics also has some disadvantages which should not be disregarded. Let us just mention that a relational semantics is overloaded with information that might be of no further interest for particular considerations. For verification purposes in a Hoare-logic style, for instance, one is mostly interested in particular satisfying regular or certain irregular states. Using a relational semantics lets the essential considerations be buried under a mountain of technicalities which are actually of no interest. These were the motivating observations for the introduction of a predicate transformer semantics.

However, relations are still common and better known so we decided to add a section concerned with the relational presentation of the results presented in this chapter. In particular a proof of a basic result is given here, too. Remember that the proposed fundamental correctness notion, the so-called "correct implementation", see Def. 2.3.1 in Chap. 2, is formulated in terms of relations and Theorems 4.3.1 resp. 4.3.2 in Chap. 4 presented equivalent characterizations. We like to show how they transfer to the inhomogeneous case. Furthermore, the proofs of Theorems 5.3.1 and 5.3.2 are still open and, thus, some point-wise reasoning is unavoidable.

Let us start with preservation of relative correctness in the case of inhomogeneous state spaces, in particular we keep the denotation of the state spaces from Sect. 5.2 so the reader should have a look at Fig. 5.1 again. Starting from (5.4) which was meant to serve as a definition we already derived a predicate transformer characterization, see Theorem 5.2.1. What is missing is a relational characterization of this notion, and this is a harder task. Letting ψ and t range over $2^{\Sigma_{\mathrm{So}}}$ and Σ_{Ti} respectively – the domain of other occurring variables should be clear from the definitions – we calculate as follows.

$$
\begin{array}{ll}
 & F_{\rho_{\mathrm{Si,Ti}}} \,;\, \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \,;\, F_{\rho_{\mathrm{So,To}}} \\
\Longleftrightarrow & \quad \{\text{Lifted order and set inclusion}\} \\
 & \forall \psi, t :\ t \in F_{\rho_{\mathrm{Si,Ti}}}(\mathsf{wrp}_A.\pi_{\mathrm{Si,So}}.\psi) :\ t \in \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}}.(F_{\rho_{\mathrm{So,To}}}(\psi)) \\
\Longleftrightarrow & \quad \{\text{Definitions of } F \text{ and } \mathsf{wrp}_A\} \\
 & \forall \psi, t : \\
 & \quad (\exists s ::\ ((s,t) \in \rho_{\mathrm{Si,Ti}} \wedge \forall \sigma_s :\ (s,\sigma_s) \in R(\pi_{\mathrm{Si,So}}) :\ \sigma_s \in \psi \cup A)) : \\
 & \quad\quad (\forall \sigma_t :\ (t,\sigma_t) \in R(\pi_{\mathrm{Ti,To}}) : \\
 & \quad\quad\quad (\sigma_t \in A \ \vee\ \exists i ::\ (i,\sigma_t) \in \rho_{\mathrm{So,To}} \wedge i \in \psi))
\end{array}
$$

$$
\begin{array}{ll}
\iff & \{\exists\text{-introduction and nesting}\} \\
& \forall \psi, t, s, \sigma_t : \ (s,t) \in \rho_{\mathrm{Si,Ti}} \ \wedge \\
& \qquad \forall \sigma_s : \ (s,\sigma_s) \in R(\pi_{\mathrm{Si,So}}) : \ \sigma_s \in \psi \cup A \ \wedge \\
& \qquad (t,\sigma_t) \in R(\pi_{\mathrm{Ti,To}}) : \\
& \qquad\quad \sigma_t \in A \ \vee \ \exists i :: \ (i,\sigma_t) \in \rho_{\mathrm{So,To}} \wedge i \in \psi \\
\iff & \{\exists\text{-introduction again and trading the range}\} \\
& \forall \psi, s, \sigma_t : \ \exists t :: \ (s,t) \in \rho_{\mathrm{Si,Ti}} \wedge (t,\sigma_t) \in R(\pi_{\mathrm{Ti,To}}) : \\
& \qquad \exists \sigma_s : \ (s,\sigma_s) \in R(\pi_{\mathrm{Si,So}}) : \ \sigma_s \notin \psi \cup A \ \vee \\
& \qquad \sigma_t \in A \ \vee \\
& \qquad \exists i :: \ (i,\sigma_t) \in \rho_{\mathrm{So,To}} \wedge i \in \psi \\
\iff & \{\text{Relational composition, range of } \sigma_s \text{ explicitly} \\
& \ \ \text{and naming conventions}\} \\
& \forall \psi, s, \sigma_t : \ (s,\sigma_t) \in \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) : \\
& \qquad \exists s' : \ (s,s') \in R(\pi_{\mathrm{Si,So}}) : \ s' \in \neg\psi \ \vee \\
& \qquad \exists \omega : \ (s,\omega) \in R(\pi_{\mathrm{Si,So}}) : \ \omega \in \Omega \setminus A \ \vee \\
& \qquad \sigma_t \in A \ \vee \\
& \qquad \exists i :: \ (i,\sigma_t) \in \rho_{\mathrm{So,To}} \wedge i \in \psi \\
\iff & \{\text{This giant step is justified below}\} \\
& \forall s, \sigma_t : \ (s,\sigma_t) \in \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) : \\
& \qquad \exists s' : \ (s,s') \in R(\pi_{\mathrm{Si,So}}) : \ (s,s') \notin R(\pi_{\mathrm{Si,So}}) \ \vee \\
& \qquad \exists \omega : \ (s,\omega) \in R(\pi_{\mathrm{Si,So}}) : \ \omega \in \Omega \setminus A \ \vee \\
& \qquad \sigma_t \in A \ \vee \\
& \qquad \exists s' :: \ (s',\sigma_t) \in \rho_{\mathrm{So,To}} \wedge (s,s') \in R(\pi_{\mathrm{Si,So}}) \\
\iff & \{\text{The first disjunct equals false, relational composition}\} \\
& \forall s, \sigma_t : \ (s,\sigma_t) \in \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) : \\
& \qquad \exists \omega : \ (s,\omega) \in R(\pi_{\mathrm{Si,So}}) : \ \omega \in \Omega \setminus A \ \vee \\
& \qquad \sigma_t \in A \ \vee \\
& \qquad (s,\sigma_t) \in R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,To}} \\
\iff & \{\text{Set-theoretic formulation}\} \\
& \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) \ \subseteq \ R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,To}} \ \cup \\
& \qquad\qquad\qquad \Sigma_{\mathrm{Si}} \times A \ \cup \\
& \qquad\qquad\qquad R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega)
\end{array}
$$

The most exciting step is of course the last but two step where we get rid of the predicate argument, see also the proof of Theorem 4.3.1. Let us justify this. We choose some arbitrary s and σ_t and firstly instantiate the upper formula with the predicate $\psi = \{s' \in \Sigma \mid (s,s') \in R(\pi_{\mathrm{Si,So}})\}$ which is the set of all regular results of $\pi_{\mathrm{Si,So}}$ started the given initial state s. Then the lower formula follows immediately. To prove that the lower formula implies the upper we assume the

lower to hold, and furthermore we choose an arbitrary predicate ψ. Now, imagine that $\forall \sigma_s : \ (s, \sigma_s) \in R(\pi_{\mathrm{Si,So}}) : \ \sigma_s \in \psi \cup A$ and that $\forall i : \ (i, \sigma_t) \in \rho_{\mathrm{So,To}} : \ i \in \neg\psi$ so we are left with showing that $\sigma_t \in A$. This is the time to exploit the premise. The first disjunct cannot hold, and so does the second because we assumed the opposite. If we assume the fourth disjunct to hold we conclude the existence of an s' which is a possible regular result of $\pi_{\mathrm{Si,So}}$ started in s related to σ_t by $\rho_{\mathrm{So,To}}$. By the first assumption above we conclude that $s' \in \psi$ whereas we also conclude that $s' \in \neg\psi$ by the second assumption. Hence, this fourth disjunct cannot hold either and we are done because the third disjunct must hold which was left to be shown.

This explanation proves the missing step and hence, in combination with Theorem 5.2.1, the theorem below. Note that it particularly generalizes Theorem 4.3.1 of Chap. 4 which is concerned with the respective situation where all state spaces are assumed to be equal. In this case the involved data representation relations are identities so they can be omitted.

Theorem 5.4.1 (PRC, inhomogeneous case). The following three conditions are equivalent.

1. (Preservation)

$$\forall \phi, \psi : \ \langle \phi \rangle \pi_{\mathrm{Si,So}} \langle \psi \rangle_A : \ \langle F_{\rho_{\mathrm{Si,Ti}}}(\phi) \rangle \pi_{\mathrm{Ti,To}} \langle F_{\rho_{\mathrm{So,To}}}(\psi) \rangle_A$$

2. (Refinement)

$$F_{\rho_{\mathrm{Si,Ti}}} \ ; \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \ ; F_{\rho_{\mathrm{So,To}}}$$

3. (Relational)

$$\begin{array}{lcl} \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) & \subseteq & R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,To}} \ \cup \\ & & \Sigma_{\mathrm{Si}} \times A \ \cup \\ & & R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \end{array}$$

□

People familiar with the classic notion of preservation of partial correctness seen from a relational perspective will recognize the latter formulation. It is said that "the diagram commutes", cf. Fig. 5.1, and in our more general setting this parlance is illustrated as follows. Consider there is a counter clockwise path – in the diagram – from the upper left corner to the lower right corner via the lower left corner. Operationally this means that program $\pi_{\mathrm{Ti,To}}$, i.e. the target program, starts in a state t which is a representation of a state of the source language, say s, and delivers an outcome, say σ_t. Then the relational formulation of Theorem 5.4.1 claims that either there is also a clockwise path from the upper left corner to the lower right corner via the upper right corner ($\pi_{\mathrm{Si,So}}$ delivers a regular result which is related to σ_t by $\rho_{\mathrm{So,To}}$), or σ_t is an accepted outcome (then we don't care about this outcome) or $\pi_{\mathrm{Si,So}}$ delivers a non-accepted outcome (then the target program need not to be relatively correct because the source program is neither). Note that some typing constraints allow this interpretation.

It is remarkable that this "commutativity" – also known under the name "L-simulation" and, though not the origin, [16] deals with notions like these – does not apply only for the classic concept of preservation of partial correctness but also for the very general case. Preservation of total correctness, for instance, is typically expressed by means of a so-called "U-simulation": Whenever $\pi_{\mathrm{Si,So}}$ starts in an initial state, say s, delivering a regular result, say t, then $\pi_{\mathrm{Ti,To}}$ delivers a regular result which is related to t by $\rho_{\mathrm{So,To}}$ if started in any of the initial states which are representations of s. Theorem 5.4.1 above shows that both, preservation of partial and preservation of total correctness, can be expressed in terms of an "L-simulation".

5.4.1 Vertical composition

Section 5.3 started with vertical composition and left open the pointwise proof of Theorem 5.3.1; this is the place to perform it. To resume, the goal was to establish

$$F_{\rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}}} \; ; \mathsf{wrp}_A.\pi_{\mathrm{Si,So}} \leq \mathsf{wrp}_A.\pi_{\mathrm{Ti,To}} \; ; F_{\rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}} \quad ,$$

c.f. (5.5), under the assumption that the translations of $\pi_{\mathrm{Si,So}}$ to $\pi_{\mathrm{Ii,Io}}$ and of $\pi'_{\mathrm{Ii,Io}}$ to $\pi_{\mathrm{Ti,To}}$ preserve relative correctness w.r.t. A and the involved data representation relations. By Theorem 5.4.1 the claim is equivalent to

$$\begin{array}{rcl} \rho_{\mathrm{Si,Ii}} \circ \rho_{\mathrm{Ii,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) & \subseteq & R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}} \;\cup \\ & & \Sigma_{\mathrm{Si}} \times A \;\cup \\ & & R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \end{array} \tag{5.13}$$

presuming that

$$\begin{array}{rcl} \rho_{\mathrm{Si,Ii}} \circ R(\pi_{\mathrm{Ii,Io}}) & \subseteq & R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}} \;\cup \\ & & \Sigma_{\mathrm{Si}} \times A \;\cup \\ & & R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{Io}} \cup \Omega) \end{array} \tag{5.14}$$

and

$$\begin{array}{rcl} \rho_{\mathrm{Ii,Ti}} \circ R(\pi_{\mathrm{Ti,To}}) & \subseteq & R(\pi'_{\mathrm{Ii,Io}}) \circ \rho_{\mathrm{Io,To}} \;\cup \\ & & \Sigma_{\mathrm{Ii}} \times A \;\cup \\ & & R(\pi'_{\mathrm{Ii,Io}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \;\; . \end{array} \tag{5.15}$$

On account of being comprehensible no further premises are assumed, instead appropriate ones will be elaborated – and of course emphasized – directly and preferably late in the actual proof. For the sake of brevity, $x \; P \; y$ is a short hand notation for $(x, y) \in P$. Furthermore, $x \; P \; y \; Q \; z$ denotes that there exists an element y of a fitting type such that $(x, y) \in P$ and $(y, z) \in Q$, or $x \; P \; y$ and $y \; Q \; z$ for short.

We start the clumsy but essential proof by taking some arbitrary x, y, u and v such that $x \; \rho_{\mathrm{Si,Ii}} \; u \; \rho_{\mathrm{Ii,Ti}} \; v \; R(\pi_{\mathrm{Ti,To}}) \; y$. In particular $u \; \rho_{\mathrm{Ii,Ti}} \; v \; R(\pi_{\mathrm{Ti,To}}) \; y$ such that

1.)$u \; R(\pi'_{\mathrm{Ii,Io}}) \; w \; \rho_{\mathrm{Io,To}} \; y$, or
2.)$y \in A$, or

3.)$u\ R(\pi'_{\mathrm{Ii,Io}})\ r$ with $r \in \Omega \setminus A$

follows from (5.15). In case 2.) we are obviously done so let us have a closer look at 1.). In order to apply the first premise, i.e. (5.14), – this seems to be the proximate step to continue – we have to seek appropriate requirements on the relationship between $\pi'_{\mathrm{Ii,Io}}$ and $\pi_{\mathrm{Ii,Io}}$. If we had $R(\pi'_{\mathrm{Ii,Io}}) \subseteq R(\pi_{\mathrm{Ii,Io}})$ we could conclude $u\ R(\pi_{\mathrm{Ii,Io}})\ w$ which would pave the way to apply (5.14). But this would be a too strong requirement because, actually, we are only interested in the initial state u and final state w in question. Therefore, we have a closer look at u's and w's domain. Obviously, it suffices to respect pairs (u, w) such that

$$u \in R_{\mathrm{in}} \stackrel{\mathrm{def}}{=} \rho_{\mathrm{Si,Ii}}(\Sigma_{\mathrm{Si}}) \cap \rho^{-1}_{\mathrm{Ii,Ti}}(R(\pi_{\mathrm{Ti,To}})^{-1}(\Sigma_{\mathrm{To}} \cup \Omega))$$

and

$$w \in R_{\mathrm{out}} \stackrel{\mathrm{def}}{=} \rho^{-1}_{\mathrm{Io,To}}(\Sigma_{\mathrm{To}})$$

because with this choice the assumption

$$R(\pi'_{\mathrm{Ii,Io}}) \cap (R_{\mathrm{in}} \times R_{\mathrm{out}}) \subseteq R(\pi_{\mathrm{Ii,Io}}) \tag{5.16}$$

allows to infer $(u, w) \in R(\pi_{\mathrm{Ii,Io}})$ and consequently $x\ \rho_{\mathrm{Si,Ii}}\ u\ R(\pi_{\mathrm{Ii,Io}})\ w$ such that

$1a.)\ x\ R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}}\ w$, or
$1b.)\ w \in A$, or
$1c.)\ x\ R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{Io}} \cup \Omega)\ w$

follows from (5.14). In case $1a.$) we are done because $x\ R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}}\ w$ and $w\ \rho_{\mathrm{Io,To}}\ y$, see 1.), implies $x\ R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}} \circ \rho_{\mathrm{Io,To}}\ y$. In particular, w is a regular state so that case $1b.$) cannot occur due to the disjointness of Σ and Ω. Case $1c.$), finally, is obvious because $x\ R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega)\ y$ follows immediately. Having finished case 1.) let us come to case 3.). Similar to the argumentation above we claim that

$$R(\pi'_{\mathrm{Ii,Io}}) \cap (R_{\mathrm{in}} \times (\Omega \setminus A)) \subseteq R(\pi_{\mathrm{Ii,Io}}) \tag{5.17}$$

such that

$3a.)\ x\ R(\pi_{\mathrm{Si,So}}) \circ \rho_{\mathrm{So,Io}}\ r$, or
$3b.)\ r \in A$, or
$3c.)\ x\ R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{Io}} \cup \Omega)\ r$

follows from (5.14). Here, case $3a.$) cannot occur due to typing constraints, $r \in \Omega$ and $\rho_{\mathrm{So,Io}} \subseteq \Sigma_{\mathrm{So}} \times \Sigma_{\mathrm{Io}}$, and neither can case $3b.$) as $r \in \Omega \setminus A$. Finally, case $3c.$) obviously implies $x\ R(\pi_{\mathrm{Si,So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega)\ y$ which completes the proof.

Putting the pieces together we just proved that, relationally spoken, the composed diagram commutes, i.e. (5.13), if the parts of which it is built commute, see (5.14) and (5.15), and if furthermore

$$R(\pi'_{\mathrm{Ii,Io}}) \cap (R_{\mathrm{in}} \times (R_{\mathrm{out}} \cup \Omega \setminus A)) \subseteq R(\pi_{\mathrm{Ii,Io}})\ . \tag{5.18}$$

In particular this proves Theorem 5.3.1, just as promised.

5.4.2 Horizontal composition

Still open is the proof of Theorem 5.3.2 which is concerned with horizontal composition, note that the naming of the state spaces has changed.

Again, the components are supposed to commute, i.e.

$$\begin{aligned} \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,T'}}) \;\subseteq\; & R(\pi_{\mathrm{Si,S'}}) \circ \rho_{\mathrm{S',T'}} \;\cup \\ & \Sigma_{\mathrm{Si}} \times A \;\cup \\ & R(\pi_{\mathrm{Si,S'}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{T'}} \cup \Omega) \end{aligned} \tag{5.19}$$

and

$$\begin{aligned} \rho'_{\mathrm{S',T'}} \circ R(\pi_{\mathrm{T',To}}) \;\subseteq\; & R(\pi_{\mathrm{S',So}}) \circ \rho_{\mathrm{So,To}} \;\cup \\ & \Sigma_{\mathrm{S'}} \times A \;\cup \\ & R(\pi_{\mathrm{S',So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \;\; , \end{aligned} \tag{5.20}$$

but this time the claim reads

$$\begin{aligned} \rho_{\mathrm{Si,Ti}} \circ R(\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}}) \;\subseteq\; & R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}}) \circ \rho_{\mathrm{So,To}} \cup \\ & \Sigma_{\mathrm{Si}} \times A \cup \\ & R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \;\; . \end{aligned} \tag{5.21}$$

We start by choosing arbitrary x, u and y such that $x \; \rho_{\mathrm{Si,Ti}} \; u \; R(\pi_{\mathrm{Ti,T'}} \bullet \pi_{\mathrm{T',To}}) \; y$. Due to the error strict definition of sequential composition of programs two basic cases have to be considered.

Firstly, assume that $x \; \rho_{\mathrm{Si,Ti}} \; u \; R(\pi_{\mathrm{Ti,T'}}) \; y$ with $y \in \Omega$. Then we are already done by (5.19) because either $x \; R(\pi_{\mathrm{Si,S'}}) \circ \rho_{\mathrm{S',T'}} \; y$ follows – which is impossible by typing constraints – or $y \in A$ resp. $x \; R(\pi_{\mathrm{Si,S'}} \bullet \pi_{\mathrm{S',So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \; y$ follows where the latter is again obtained by applying the "error-strict" definition of sequential composition.

Hence, let us come to the more interesting case of a regular intermediate state and assume $x \; \rho_{\mathrm{Si,Ti}} \; u \; R(\pi_{\mathrm{Ti,T'}}) \; v \; R(\pi_{\mathrm{T',To}}) \; y$, where v is supposed to be a regular state. In particular $x \; \rho_{\mathrm{Si,Ti}} \; u \; R(\pi_{\mathrm{Ti,T'}}) \; v$ which, by (5.19), allows to conclude

1.)$x \; R(\pi_{\mathrm{Si,S'}}) \; w \; \rho_{\mathrm{S',T'}} \; v$, or
2.)$v \in A$, or
3.)$x \; R(\pi_{\mathrm{Si,S'}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{T'}} \cup \Omega) \; v$.

Here, case 2.) cannot take place because v is supposed to be a regular state, and case 3.) lets us finish using the same arguments as before. The remaining case is 1.) and here we are in search of sufficient requirements on the relationship between $\rho_{\mathrm{S',T'}}$ and $\rho'_{\mathrm{S',T'}}$ in order to proceed. Let us assume, for the moment, that $w \; \rho'_{\mathrm{S',T'}} \; v$. Then also $w \; \rho'_{\mathrm{S',T'}} \; v \; R(\pi_{\mathrm{T',To}}) \; y$ which allows to apply (5.20) and thus to conclude

1a.) $w \; R(\pi_{\mathrm{S',So}}) \circ \rho_{\mathrm{So,To}} \; y$, or
1b.) $y \in A$, or
1c.) $w \; R(\pi_{\mathrm{S',So}})^{-1}(\Omega \setminus A) \times (\Sigma_{\mathrm{To}} \cup \Omega) \; y$.

In case 1*b*.) we are done, case 1*c*.) follows the lines from above and, finally, composing 1.) and 1*a*.) yields $x\ R(\pi_{\mathrm{Si},\mathrm{S}'})\ w\ R(\pi_{\mathrm{S}',\mathrm{So}}) \circ \rho_{\mathrm{S}',\mathrm{T}'}\ y$ which completes the proof.
What remains is to find sufficient requirements which justify the assumption $w\ \rho'_{\mathrm{S}',\mathrm{T}'}\ v$. By 1.) we know that $w\ \rho_{\mathrm{S}',\mathrm{T}'}\ v$ but claiming $\rho_{\mathrm{S}',\mathrm{T}'} \subseteq \rho'_{\mathrm{S}',\mathrm{T}'}$, again, seems too strong so it is appropriate to have a closer look at the domains instead. Here, the two candidates serving for a restricting set are

$$R_{\mathrm{in}} \stackrel{\mathrm{def}}{=} R(\pi_{\mathrm{Si},\mathrm{S}'})(\Sigma_{\mathrm{Si}}) \tag{5.22}$$

and

$$R_{\mathrm{out}} \stackrel{\mathrm{def}}{=} R(\pi_{\mathrm{Ti},\mathrm{T}'})(\rho_{\mathrm{Si},\mathrm{Ti}}(\Sigma_{\mathrm{Si}})) \cap R(\pi_{\mathrm{T}',\mathrm{To}})^{-1}(\Sigma_{\mathrm{To}} \cup \Omega) \tag{5.23}$$

because $w \in R_{\mathrm{in}}$ and $v \in R_{\mathrm{out}}$ with this choice. The very final premise is thus, putting the pieces together,

$$\rho_{\mathrm{S}',\mathrm{T}'} \cap (R_{\mathrm{in}} \times R_{\mathrm{out}}) \subseteq \rho'_{\mathrm{S}',\mathrm{T}'}\ , \tag{5.24}$$

which suffices to guarantee that the horizontally composed diagram commutes, i.e. (5.21), if the parts do of which it consists, cf. (5.14) and (5.15). In the predicate transformer setting this proves Theorem 5.3.2 as required.

We like to close with the following remarks. Some of the previously visited cases were obvious by typing constraints. Indeed, the simplified view on data representations as relations on regular states allows this proceeding. It might be kept for more realistic to relate also erroneous outcomes but this would entail yet some more modifications without bringing any further enlightenments. Let us just mention that, depending on how far-reaching the modifications are, the involved relations and their composition must be "error-set-strict" in the sense that they must not mix accepted and rejected outcomes (one would, for instance, use a one-to-one correspondence). Thus, the data representation relations would depend on the choice of A (or would behave like identities) so our simplified view has its right, too.

5.5 Remarks

The motivating needs for inserting this chapter were already discussed in the introduction: On account of justifying the subtitle of the present thesis we showed how reasoning in terms of wrp-transformers gets along with the practical demands of building verified compilers in a modular fashion. In particular, the notion of PRC was extended to the realistic scenario with inhomogeneous state spaces, cf. Theorem 5.4.1, and it was shown under which circumstances a vertically or sequentially composed compiler inherits the relative correctness preservation properties from its components, see Theorems 5.3.1 and 5.3.2.

Finally, we like to append the below intermezzo in order to prevent from some misunderstandings that might have popped up in the present chapter. It does not present any further insights, on the contrary, most of the things reported there are rather trivial, but the experience shows that they are nevertheless worth mentioning.

5.5.1 Effects of the use of monotonicity

The wrp-family was basically introduced to be much more abstract in several respects. Firstly, a relational semantics, which is supposed to be the ultimate reference and also the starting point for their derivation, is overloaded with information that might be of no further interest what some actual verification purposes concerns. Partitioning the set of irregular outcomes into ones to be accepted resp. to be rejected and generally reasoning in terms of sets of states (predicates) instead of single states turned out to be quite more manageable and handy. Secondly, the space of predicate transformers, which is a complete lattice, allows to argue strictly algebraically and to benefit from the lifted order. And even more, as will be shown in Chap. 6, the space of relations corresponds to a subset of the space of monotonic predicate transformers so predicate transformers can be taken into account which do not comply with relations; e.g. lower adjoints of universally conjunctive predicate transformers.

However, the reader might wonder why Theorems 5.3.1 and 5.3.2 turned out to be improvable in a purely algebraic manner in the (richer) space of predicate transformers and we like to comment on this briefly.

The cause for this dilemma lies in the exploitation of monotonicity; in this case the monotonicity of the composition operator ';' in both arguments. Let us discuss the problem roughly but sufficiently detailed by means of the following illustrative example. Suppose we want to prove resp. are in search of requirements implying

$$L \,;\, F \,;\, R \leq L \,;\, G \,;\, R \ , \tag{5.25}$$

where F and G are supposed to be predicate transformers which are surrounded by predicate transformers L and R. A typical algebraic step would be to exploit the monotonicity of the composition operator and obtain

$$L \,;\, F \leq L \,;\, G \quad \text{, or} \quad F \,;\, R \leq G \,;\, R \quad \text{, or even} \quad F \leq G$$

as a sufficient requirement establishing (5.25). If, for instance, $F \leq G$ is obvious or universally valid we are done but it is more interesting to assume that certain assumptions have to be made in order to guarantee (5.25). The next step would be to start a pointwise calculation which hopefully would reveal the searched requirements. Now, it makes essential difference from which of the three inequations we start this calculation. Depending on the first, the second or the third version either the information kept in R, in L or even in both, L and R, gets lost; F's and G's right or/and left context – which is of interest in the very end – has vanished. We are actually interested in showing (5.25) which operationally means that F resp. G is fed with input obtained by R, and F's resp. G's output serves for an input to L. From this viewpoint it is obvious that one gets stronger requirements on F and G establishing (5.25) because the goal is also strengthened.[7] If restricting context is missing one has to add stronger premises in order to balance

[7] Consider, for instance, one seeks for a set of solutions of the inequality $ax + b \leq c$ over the natural numbers. If $d \leq c$ then it suffices to consider the "simpler" inequality $ax + b \leq d$ but, of course, one will never get the weakest set of solutions which is the greatest in this case.

its disappearance. Moreover, the weakest requirements can only be found if all non-redundant information is available.

In particular, not the predicate transformers are to be blamed for this sad state of affairs but solely monotonicity. Similar problems emerge in different scenarios which are also equipped with an order. Consider, for instance, that (5.13) is to be shown in the abstract, i.e. point-free, relation algebra. A first step would be to exploit monotonicity in the sense that the left hand side of (5.13) is estimated by $\rho_{\mathrm{Ii,Ti}}$ composed with the right hand side of (5.15). Again, $\rho_{\mathrm{Si,Ii}}$ has vanished and, thus, cannot appear in the searched requirements.

We like to stress that the predicate transformer versions of the goals for horizontal and vertical composition, (5.5) and (5.9), could of course be pointwisely shown by unrolling the definitions. But as predicate transformers are defined on base of relations this proceeding has no advantage over the one presented here. On the contrary, there is one particular reason why starting from the relational setting might be preferable or at least more natural. Reasoning in terms of predicate transformers means to reason in a well-defined framework. To compare two predicate transformers w.r.t. the order '$\leq$' one typically and firstly checks if the usage of this symbol is allowed at all, i.e. if the two predicate transformers are of appropriate type (if the two predicate transformers are members of the same lattice). This is different for relational inclusions because the order '$\subseteq$' is just the set-inclusion and one usually feels quite more familiar with set theory than with lattice theory and monoids. To picture it more flowery: The fact that it might be kept for preferable to reason pointwisely for relations is due to history; maybe one should do the same for predicate transformers.

Finally, and this is a fetch-ahead from the next chapter, each wrp-transformer corresponds to a relation such that one can always go down to the relational level – even if the wrp-transformers are defined axiomatically – in order to calculate pointwisely. Obtained results can be lifted back to the predicate transformer level without loss of information. Such detours are typical and ubiquitous in mathematics; if, for instance, something seems improvable from the group-theory axioms one descends to the underlying model which is of actual interest and tries a direct proof.

6. Theoretical Aspects of wrp

Before going into details of more practice-oriented program and translation verification by means of wrp-transformers it is useful to have a closer look at the algebraic properties they enjoy. We concentrate on the basic ones and feel guided by Dijkstra's and Scholten's [18] where similar rules are postulated to act as so-called *healthiness conditions* which the two transformers wp and wlp have to satisfy in order to be reasonable for describing semantics of implementable imperative programs. It turns out that the extreme instances of wrp, namely $\mathsf{wrp}_{\emptyset}$ and wrp_{Ω}, do satisfy them so they can be taken for adequate in their restricted world where all kinds of failures are identified.

However, the wrp-family was introduced distinguish between the variety of errors. Thus it is not astonishing that the rules presented here are more general but it is also nice to see that they show certain similarities to the healthiness conditions presented in [18]. Colloquially speaking, one can keep the wrp-rules for generalized versions of the ones by Dijkstra and Scholten; this is just due to the more careful differentiation between erroneous outcomes.

Having composed the basic rules and after a closer look at the mentioned healthiness conditions the question is discussed if some of the derived rules could also serve for an axiomatic definition of the family of wrp-transformers. We choose the generalized versions of Dijkstra's and Scholten's healthiness conditions and concentrate on questions of expressiveness. To be precise, we show that reasoning in the world of (total, univalent, functional) relations is as expressive as reasoning in the world of wrp-transformers satisfying two (resp. three, four, five) certain laws. Here we follow the lines of Hesselink's text-book [31] where some – not all – corresponding results for the simplified world can be found.

6.1 Basic Properties of wrp

Not surprisingly, each wrp-transformer is conjunctive, thus monotonic, in both the failure-outcome and the predicate argument which is due to the definition via a universal quantification over a disjunction. If, operationally speaking, each computation delivers a regular result satisfying ϕ or an irregular outcome contained in A, and if furthermore each computation also delivers a regular result satisfying ψ or an irregular outcome contained in B, then each computation delivers a regular result satisfying $\phi \cap \psi$ or an irregular outcome contained in $A \cap B$. The other direction is even more obvious but, nevertheless, the following Lemma deserves a proof.

Lemma 6.1.1 (Generalized pairing condition). For all $A, B \subseteq \Omega$ and all $\phi, \psi \in \mathit{Pred}$:

$$\mathsf{wrp}_A.\pi.\phi \ \cap \ \mathsf{wrp}_B.\pi.\psi \ = \ \mathsf{wrp}_{A \cap B}.\pi.(\phi \cap \psi) \ .$$

Proof. For all $s \in \Sigma$:

$$\begin{array}{ll} & s \in \mathsf{wrp}_A.\pi.\phi \cap \mathsf{wrp}_B.\pi.\psi \\ \iff & \quad \{\text{Definition of } \mathsf{wrp}\} \\ & \forall \sigma : \ (s,\sigma) \in R(\pi) : \ s \in \phi \cup A \ \wedge \\ & \forall \sigma : \ (s,\sigma) \in R(\pi) : \ s \in \psi \cup B \\ \iff & \quad \{\forall \text{ distributes over } \wedge\} \\ & \forall \sigma : \ (s,\sigma) \in R(\pi) : \ s \in (\phi \cup A) \cap (\psi \cup B) \\ \iff & \quad \{\text{Distribute, } \Sigma \cap \Omega = \emptyset\} \\ & \forall \sigma : \ (s,\sigma) \in R(\pi) : \ s \in (\phi \cap \psi) \cup (A \cap B) \\ \iff & \quad \{\text{Definition of } \mathsf{wrp}\} \\ & s \in \mathsf{wrp}_{A \cap B}.\pi.(\phi \cap \psi) \ . \end{array}$$

□

Simple as it is, this lemma is fundamental because of its plenty of consequences and applications. But before going into details we observe that two extreme instances of wrp are universally conjunctive.

Lemma 6.1.2 (srf and sp). The functions

$$\mathsf{wrp}_\bullet.\pi.\mathtt{true} \in (2^\Omega \to \mathit{Pred}) \quad \text{and} \quad \mathsf{wrp}_\Omega.\pi \in \mathit{PTrans}$$

are universally conjunctive.

Proof. Let us start with the better known second claim. For all $\phi, \psi \in \mathit{Pred}$ we calculate

$$\begin{array}{ll} & \phi \subseteq \mathsf{wrp}_\Omega.\pi.\psi \\ \iff & \quad \{\text{Set inclusion and definition of } \mathsf{wrp}\} \\ & \forall s : \ s \in \phi : \ (\forall \sigma : \ (s,\sigma) \in R(\pi) : \ \sigma \in \psi \vee \sigma \in \Omega) \\ \iff & \quad \{\text{Nesting}\} \\ & \forall s, \sigma : \ s \in \phi \wedge (s,\sigma) \in R(\pi) : \ \sigma \in \psi \vee \sigma \in \Omega \\ \iff & \quad \{\text{Trading the range, } \sigma \notin \Omega \iff \sigma \in \Sigma\} \\ & \forall s, \sigma : \ s \in \phi \wedge (s,\sigma) \in R(\pi) \wedge \sigma \in \Sigma : \ \sigma \in \psi \\ \iff & \quad \{\exists\text{-Introduction, } \exists \text{ and } \wedge\} \\ & \forall \sigma : \ (\exists s : \ (s,\sigma) \in R(\pi) : \ s \in \phi) \wedge \sigma \in \Sigma : \ \sigma \in \psi \\ \iff & \quad \{\text{Naming convention, define } \mathsf{sp}.\pi \in \mathit{PTrans} \text{ by} \\ & \qquad \mathsf{sp}.\pi.\phi \stackrel{\text{def}}{=} \{s' \in \Sigma \mid \exists s : \ (s,s') \in R(\pi) : \ s \in \phi\}\} \\ & \forall s' : \ s' \in \mathsf{sp}.\pi.\phi : \ s' \in \psi \\ \iff & \quad \{\text{Set inclusion}\} \\ & \mathsf{sp}.\pi.\phi \subseteq \psi \ . \end{array}$$

Hence, $\mathsf{wrp}_\Omega.\pi$ has a lower adjoint, namely $\mathsf{sp}.\pi$ defined by

$$\mathsf{sp}.\pi.\phi \stackrel{\text{def}}{=} \{s' \in \Sigma \mid \exists s :\ (s,s') \in R(\pi) :\ s \in \phi\}\ ,$$

and is thus universally conjunctive, see Sect. 3.2. Analogously, defining $\mathsf{srf}.\pi \in (\mathit{Pred} \to 2^\Omega)$ by

$$\mathsf{srf}.\pi.\phi \stackrel{\text{def}}{=} \{\omega \in \Omega \mid \exists s :\ (s,\omega) \in R(\pi) :\ s \in \phi\}$$

for any $\phi \in \mathit{Pred}$ allows a similar proof of the first claim (note that the powerset 2^Ω also forms a complete lattice). Thus, $(\mathsf{sp}.\pi, \mathsf{wlp}.\pi)$ and $(\mathsf{srf}.\pi, \mathsf{wrp}_\bullet.\pi.\mathtt{true})$ are Galois connections.
□

We preferred the above proof because in [18] the lower adjoint $\mathsf{sp}.\pi$ of $\mathsf{wrp}_\Omega.\pi = \mathsf{wlp}.\pi$ is introduced under the name *strongest postcondition* predicate transformer. Applied to a precondition ϕ it yields the smallest set of regular states that can be reached by a π-computation started in initial states satisfying ϕ. Analogously the lower adjoint $\mathsf{srf}.\pi$ of $\mathsf{wrp}_\bullet.\pi.\mathtt{true}$ could be called the *strongest reachable failure* transformer – note that this is not a predicate transformer – because, applied to a precondition ϕ, $\mathsf{srf}.\pi.\phi$ is the smallest set of irregular outcomes that can be reached by a computation under control of π started in states satisfying ϕ. The transformer $\mathsf{srf}.\pi$ can even be of practical interest: Proving absence of failures in $\Omega \setminus A$ reachable from ϕ, i.e showing that all erroneous outcomes reachable from ϕ are contained in A, in symbols $\langle\phi\rangle\pi\langle\mathtt{true}\rangle_A$, is equivalent to showing $\mathsf{srf}.\pi.\phi \subseteq A$. It is nice to see that our more widespread setting also has a counterpart in the failure argument.

A trivial consequence is the following. If a π-computation can start at all – note that this might not be the case as the underlying relational semantics might not be total – then it delivers an outcome. Obvious as it is, to stimulate an algebraic feeling we like to mention it anyhow.

Corollary 6.1.1 (Starting computations yield outcomes).

$$\mathsf{wrp}_\Omega.\pi.\mathtt{true} = \mathtt{true}$$

Proof. Use universal conjunctivity of $\mathsf{wrp}_\Omega.\pi$ in

$$\mathtt{true} \subseteq \mathsf{wrp}_\Omega.\pi.\mathtt{true} \quad\Longleftrightarrow\quad \mathsf{sp}.\pi.\mathtt{true} \subseteq \mathtt{true}\ ,$$

and the latter as well as the other inclusion obviously holds.
□

Yet another simple consequence is the below counterpart of the 'Best of both worlds' law known from predicate logic. If all π-computations yield outcomes which satisfy ψ or which are contained in A but if there is also a π-computation yielding outcomes which satisfy ϕ or which are contained in B then there must be a computation which yields outcomes satisfying the conjunction of ψ and ϕ or which are contained in the intersection of A and B.

Lemma 6.1.3 (Best of both worlds).

$$\mathsf{wrp}_A.\pi.\psi \ \cap\ \neg\mathsf{wrp}_{\Omega\setminus B}.\pi.\neg\phi \ \subseteq\ \neg\mathsf{wrp}_{\Omega\setminus(A\cap B)}.\pi.\neg(\psi \cap \phi)$$

Proof.

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.\pi.\psi \cap \neg\mathsf{wrp}_{\Omega\setminus B}.\pi.\neg\phi \subseteq \neg\mathsf{wrp}_{\Omega\setminus(A\cap B)}.\pi.\neg(\psi\cap\phi) \\
\Longleftarrow & \quad \{\text{Shunt twice}\} \\
 & \mathsf{wrp}_A.\pi.\psi \cap \mathsf{wrp}_{\Omega\setminus(A\cap B)}.\pi.\neg(\psi\cap\phi) \subseteq \mathsf{wrp}_{\Omega\setminus B}.\pi.\neg\phi \\
\Longleftarrow & \quad \{\text{Distribute and monotonicity,} \\
 & \quad\ \ \text{see generalized pairing condition}\} \\
 & \text{True}
\end{array}
$$

□

The next inconspicuously looking lemma – it hardly deserves this name – is a central one as it is also a healthiness condition in [18]. It states that, colloquially speaking, a computation can never terminate regularly in a state satisfying `false`. Notice that the following rules heavily depend on the relational semantics $R(\pi)$ being total.

Lemma 6.1.4 (Law of the excluded miracle).

$$R(\pi) \text{ is total} \quad\Longleftrightarrow\quad \mathsf{wrp}_\emptyset.\pi.\mathtt{false} = \mathtt{false}$$

Proof.

$$
\begin{array}{ll}
 & \mathsf{wrp}_\emptyset.\pi.\mathtt{false} \subseteq \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\} \\
 & \forall s :\ s \in \mathsf{wrp}_\emptyset.\pi.\mathtt{false} :\ s \in \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Trading the range, } s \in \Sigma \text{ by naming convention}\} \\
 & \forall s ::\ s \notin \mathsf{wrp}_\emptyset.\pi.\mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Definition of } \mathsf{wrp}\} \\
 & \forall s ::\ (\exists\sigma :\ (s,\sigma) \in R(\pi) :\ \sigma \notin \mathtt{false}) \\
\Longleftrightarrow & \quad \{\sigma \notin \mathtt{false} \text{ holds obviously}\} \\
 & \forall s ::\ (\exists\sigma ::\ (s,\sigma) \in R(\pi)) \\
\Longleftrightarrow & \quad \{\text{Definition of totality (2.2)}\} \\
 & R(\pi) \text{ is total}
\end{array}
$$

□

The name of this law is due to [18] but let us skip questions what similarities concerns for a moment.

The predicate $\mathsf{wrp}_A.\pi.\psi$ characterizes the set of initial states from which outcomes contained in A or ψ are unavoidable. The predicate $\neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi)$ on the other side is the set of initial states from which outcomes contained in A or ψ are merely possible. Hence, the first set should be contained in the second, but note again that this is only the case if π is guaranteed to have at least one outcome, i.e. if $R(\pi)$ is total. The converse, however, should only be valid for deterministic programs. By the way, the third point below serves for the definition of determinism in [18].

Lemma 6.1.5 (wrp and determinism).

1. For all $A \subseteq \Omega$ and $\psi \in \mathit{Pred}$: $R(\pi)$ is total if and only if
$$\mathsf{wrp}_A.\pi.\psi \;\subseteq\; \neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi) \;.$$
2. $R(\pi)$ is univalent if and only if for all $A \subseteq \Omega$ and $\psi \in \mathit{Pred}$:
$$\mathsf{wrp}_A.\pi.\psi \;\supseteq\; \neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi) \;.$$
3. $R(\pi)$ is a function if and only if for all $A \subseteq \Omega$ and $\psi \in \mathit{Pred}$:
$$\mathsf{wrp}_A.\pi.\psi \;=\; \neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi) \;.$$

Proof. Let us start with 1, so suppose given $A \subseteq \Omega$ and $\psi \in \mathit{Pred}$. Then

$$\begin{array}{ll}
 & \mathsf{wrp}_A.\pi.\psi \;\subseteq\; \neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi) \\
\Longleftrightarrow & \quad \{\text{Shunting}\} \\
 & \mathsf{wrp}_A.\pi.\psi \cap \mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi \subseteq \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Generalized pairing condition}\} \\
 & \mathsf{wrp}_{\emptyset}.\pi.\mathtt{false} \subseteq \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Law of the excluded miracle, Lemma 6.1.4}\} \\
 & R(\pi) \text{ is total} \;.
\end{array}$$

Proving the second claim is harder:

$$\begin{array}{ll}
 & R(\pi) \text{ is univalent} \\
\Longleftrightarrow & \quad \{\text{Definition (2.3)}\} \\
 & \forall s, \sigma, \sigma' :\; (s,\sigma) \in R(\pi) \wedge (s,\sigma') \in R(\pi) :\; \sigma = \sigma' \\
\Longleftrightarrow & \quad \{\text{“}\Longrightarrow\text{” is obvious,} \\
 & \qquad \text{for “}\Longleftarrow\text{” choose } \psi = \{\sigma\} \text{ resp. } A = \{\sigma\}\} \\
 & \forall s, \sigma, \sigma', A, \psi :\; (s,\sigma) \in R(\pi) \wedge (s,\sigma') \in R(\pi) : \\
 & \qquad\qquad (\sigma \in \Sigma \wedge \sigma' \in \Sigma \wedge (\sigma \in \neg\psi \vee \sigma' \in \psi)) \;\vee \\
 & \qquad\qquad (\sigma \in \Omega \wedge \sigma' \in \Omega \wedge (\sigma \notin A \vee \sigma' \in A)) \\
\Longleftrightarrow & \quad \{\text{Distribute the claim}\} \\
 & \forall s, \sigma, \sigma', A, \psi :\; (s,\sigma) \in R(\pi) \wedge (s,\sigma') \in R(\pi) : \\
 & \qquad\qquad (\sigma \in \neg\psi \vee \sigma \in \Omega \setminus A) \vee (\sigma' \in \psi \vee \sigma' \in A) \\
\Longleftrightarrow & \quad \{\text{Unnesting, } \forall \text{ distributes over } \vee\} \\
 & \forall s, A, \psi ::\; (\forall \sigma :\; (s,\sigma) \in R(\pi) :\; \sigma \in \neg\psi \vee \sigma \in \Omega \setminus A) \;\vee \\
 & \qquad\qquad (\forall \sigma :\; (s,\sigma) \in R(\pi) :\; \sigma \in \psi \vee \sigma \in A) \\
\Longleftrightarrow & \quad \{\text{Definition of wrp}\} \\
 & \forall s, A, \psi ::\; s \in \mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi \;\vee\; s \in \mathsf{wrp}_A.\pi.\psi \\
\Longleftrightarrow & \quad \{\text{Trading the range}\} \\
 & \forall s, A, \psi :\; s \in \neg\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi :\; s \in \mathsf{wrp}_A.\pi.\psi \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\} \\
 & \forall A, \psi ::\; \neg\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi \subseteq \mathsf{wrp}_A.\pi.\psi \;.
\end{array}$$

Finally, 3 is a consequence of 1 and 2.
□

Keeping in mind that $\mathsf{wrp}_{\Omega} = \mathsf{wlp}$ and $\mathsf{wrp}_{\emptyset} = \mathsf{wp}$ we finish our exploration with the following little summary.

Corollary 6.1.2 (Derived healthiness conditions).

1. $\mathsf{wp}.\pi.\psi \;=\; \mathsf{wlp}.\pi.\psi \cap \mathsf{wp}.\pi.\mathtt{true}$ (Pairing condition)
2. $\mathsf{wp}.\pi.\mathtt{false} \;=\; \mathtt{false}$ if $R(\pi)$ is total. (Excluded miracle)
3. $\mathsf{wlp}.\pi$ is universally conjunctive and $\mathsf{wp}.\pi$ is positively conjunctive.
4. $\mathsf{wrp}_A.\pi.\psi \;=\; \neg(\mathsf{wrp}_{\Omega\setminus A}.\pi.\neg\psi)$ if π is deterministic.

□

It is this little collection of extreme instances of our wrp laws we like to discuss briefly. Laws 1 to 3 are exactly those ones Dijkstra and Scholten *postulated axiomatically* in [18] and which have to be satisfied by the pair of transformers wlp and wp in order to adequately model the semantics of implementable programs. (Of course the totality-requirement in 2 is not present because from their axiomatic point of view there does not exist an underlying relational semantics acting as a base for a derivation.) As those laws are postulated terms they have no direct operational background and are, thus, only verbally justified in [18].[1] We like to recall those explanations in our own vocabulary.
First of all, a program is totally correct if it is partially correct and if it terminates regularly for each initial state. In terms of predicate transformers this is expressed by the pairing condition.
Furthermore, no program π can terminate regularly in a final state satisfying the predicate `false`; in other words there are no initial states from which a π-computation terminates regularly in a state satisfying `false` because this indeed would establish a miracle (this is where the name of law 2 resp. Lemma 6.1.4 stems from). But we observe that this explanation is only valid if each program can start in every initial state; this argument is also present in [18]. Otherwise, a program might even have no outcome for some initial states and exactly this set contradicts the argumentation (hence the restriction to *implementable* programs). From this point of view it should be demanded that an initially given relational semantics is total but in practice this requirement is barely needed so we desisted from doing so.
The remaining distribution properties are – though in other words – motivated as follows. Consider a non-empty set of predicates Ψ and an arbitrary precondition ϕ. Then program π is totally correct w.r.t. ϕ and $\bigcap\Psi$ if and only if, for all $\psi \in \Psi$, it is totally correct w.r.t. ϕ and ψ. Hence, $\mathsf{wp}.\pi$ should be positively conjunctive. The equivalence does not hold in the case $\Psi = \emptyset$ because $\bigcap\emptyset = \mathtt{true}$ such that the left hand side expresses that π terminates regularly for every initial state satisfying

[1] To be fair and complete, there is an own chapter devoted to operational considerations (Chap. 10) but this includes possible implementations of some concrete commands, e.g. conditionals and mainly loops. It is discussed – and even proved on an abstract level – that those implementations satisfy the healthiness conditions but this is done without an underlying operational or relational semantics and the actual justification for the choice of the healthiness conditions is only motivated briefly.

ϕ whereas the right hand side holds for trivial reasons. However, the mentioned equivalence remains valid if one restricts attention to partially correct programs π; this explains the demand of $\mathsf{wlp}.\pi$ being universally conjunctive.
We added 4 to stress that this equality serves for the definition of deterministic programs in [18]. As seen in the proof of Lemma 6.1.5 one inclusion follows from the law of the excluded miracle which is a postulate so the other inclusion is the actual definition.

Our wrp transformers on the other side are *derived* terms and Corollary 6.1.2 above, thus, tells us that the extreme instances of wrp satisfy the healthiness conditions if the underlying relational semantics $R(\pi)$ is total. In this sense wrp_Ω and $\mathsf{wrp}_\emptyset$ are adequate for reasoning about semantics in a simplified world where all kinds of irregular outcomes are identified. However, the family of wrp-transformers was introduced as a remedy for this restricted view. So it might be interesting to follow Dijkstra's and Scholten's lines and *postulate* some laws from which we think they should be satisfied by a family of relativized predicate transformers in order to adequately model semantics in the richer world of more than just one failure outcome. Of course we do not want to solely justify, in fact we want to prove them to be adequate. This is what the next section is about.

6.2 An Axiomatic View

In Sect. 4.3 we saw that reasoning in terms of wrp is *at least* as expressive as reasoning in terms of a relational semantics in the following sense. Given $R(\pi)$ we can derive the family of wrp-transformers (Def. 4.3.2), and we can always transform results back to the relational setting (Lemma 4.3.2). In particular the transformation of $R(\pi)$ to the family of wrp-transformers preserves all information kept in the initial $R(\pi)$ because otherwise we could regain lost information in miraculous ways. We like to sum up and formulate this little observation as follows.

Lemma 6.2.1. A relativized predicate transformer semantics is at least as expressive as a relational semantics.
□

Yet another consequence is that each relation $R(\pi)$ corresponds to a family of *conjunctive* predicate transformers. The framework we are actually working in is the space of *monotonic* predicate transformers which is of course a richer space because each conjunctive transformer is monotonic but not vice versa. So the question might arise how to restrict the set *PTrans* of monotonic predicate transformers to wrp_A-transformers in an abstract fashion, i.e. without an underlying relational semantics, in such ways that a relational semantics in the shape of $R(\pi)$ can be derived without any loss of information. Worded differently, which properties to choose, i.e. to keep for axioms, in order to be *at most* as expressive as reasoning in terms of a relational semantics, and this is the story told here.

Let us turn directly towards our goal. Motivated by the previous section – and consequently by [18] – we choose the following laws from which we hope

they serve it: Restricting the set of predicate transformers to those transformers which correspond to a relation carrying the same information as the axiomatically postulated predicate transformer. (Remember to forget all about an initially given relational semantics, what follows has no operational background!)

Definition 6.2.1 (Axiomatic wrps). If the function

$$\mathsf{xwrp} \in (2^{\Omega} \to (\Pi \to PTrans))$$

satisfies the axioms

[AX1] For all $A, B \subseteq \Omega$ and $\phi, \psi \in Pred$:

$$\mathsf{xwrp}.A.\pi.\phi \cap \mathsf{xwrp}.B.\pi.\psi \;=\; \mathsf{xwrp}.(A \cap B).\pi.(\phi \cap \psi) \;\text{, and}$$

[AX2] The functions

$$\mathsf{xwrp}.\Omega.\pi \in PTrans \quad \text{and} \quad \mathsf{xwrp}.\bullet.\pi.\mathtt{true} \in (2^{\Omega} \to Pred)$$

are universally conjunctive,

for a fixed π then $\mathsf{xwrp}.A.\pi$ is called an *axiomatic weakest relative precondition predicate transformer* for each $A \subseteq \Omega$. The so-called *axiomatic law of the excluded miracle* holds for π if

[Excluded Miracle] $\mathsf{xwrp}.\emptyset.\pi.\mathtt{false} \;=\; \mathtt{false}$.

In the case that

[Univalence] $\neg\mathsf{xwrp}.(\Omega \setminus A).\pi.\neg\psi \;\supseteq\; \mathsf{xwrp}.A.\pi.\psi$

holds for all $A \subseteq \Omega$ and $\psi \in Pred$, program π is said to be *axiomatically univalent*. Finally, π is called *axiomatically deterministic* if

[Determinism] $\neg\mathsf{xwrp}.(\Omega \setminus A).\pi.\neg\psi \;=\; \mathsf{xwrp}.A.\pi.\psi$

holds for all $A \subseteq \Omega$ and $\psi \in Pred$.
□

A closer look at Lemma 6.1.1 and Lemma 6.1.2 lets us agree that the derived weakest relative precondition predicate transformer $\mathsf{wrp} \in (2^{\Omega} \to (\Pi \to PTrans))$ meets axioms [AX1] and [AX2]. Furthermore, $\mathsf{wrp}_{\emptyset}.\pi$ satisfies the law of the excluded miracle, thus [Excluded Miracle], iff the underlying relational semantics $R(\pi)$ is total, cf. Lemma 6.1.4. Property [Univalence] is also satisfied for $\mathsf{wrp}_A.\pi$ iff the underlying relation is univalent, and hence law [Determinism] is satisfied if $R(\pi)$ is a function, i.e. if π is deterministic in the relational sense of Sect. 2.1.

One could call the laws *sound* in some sense, because each result obtained by calculating using xwrp can also be obtained by a calculation using solely wrp, just replace each occurrence of xwrp by wrp. However, we avoid a further definition and refuse from going into details because each soundness result would scream for a completeness result and this would be far beyond our scope of presentation (but maybe a topic for future research, see the conclusion). Instead we concentrate on the question whether reasoning in terms of transformers satisfying the laws [AX1] and [AX2] (and [Excluded Miracle], [Univalence] or [Determinism]) is as

expressive as reasoning with (total, univalent or functional) relations in the shape of $R(\pi)$.

The first step is to derive a relational semantics, to avoid confusion say $D(\pi)$, from initially given xwrp-transformers. So suppose given a family of xwrp-transformers for an arbitrary $\mathsf{xwrp} \in (2^\Omega \to (\Pi \to PTrans))$ and a program $\pi \in \Pi$ satisfying the axioms [AX1] and [AX2] of definition 6.2.1. From Chap. 4, see Lemma 4.3.2, we know an auspicious candidate for a relational semantics to be derived. Keeping in mind that as yet a relational semantics is not given, xwrp is a postulated term, we decide to derive $D(\pi)$ from xwrp by[2]

$$D(\pi) \stackrel{\text{def}}{=} \{(s, s') \mid s \in \neg\mathsf{xwrp}_\Omega.\pi.\neg s'\} \cup \{(s, \omega) \mid s \in \neg\mathsf{xwrp}_{\Omega\setminus\omega}.\pi.\mathtt{true}\} \quad .(6.1)$$

Note that, just like in Lemma 4.3.2, the entire family of xwrp-transformers for the given π is needed.

The predicate transformer $\mathsf{wrp}_A.\pi$ is now derived from this – itself derived – relation $D(\pi)$ analogously to Def. 4.3.2,[3] i.e. for all $A \subseteq \Omega$, $\psi \in Pred$ and $s \in \Sigma$:

$$s \in \mathsf{wrp}_A.\pi.\psi \stackrel{\text{def}}{\Longleftrightarrow} \forall\sigma : (s, \sigma) \in D(\pi) : \sigma \in \psi \cup A \quad .$$

As mentioned before we would be quite convinced that the transformation from xwrp to $D(\pi)$ preserves all information kept in the initially given family of xwrp-transformers if we could show $\mathsf{xwrp}.A.\pi = \mathsf{wrp}_A.\pi$ for all A. Otherwise we could regain information from the derived relational semantics by mystery though information got lost during its derivation. Indeed, this is provable.

Lemma 6.2.2 (Regaining xwrp).

$$\mathsf{xwrp}.A.\pi = \mathsf{wrp}_A.\pi \quad \text{for all } A \subseteq \Omega \quad .$$

Proof. We start our considerations with the special case $A = \Omega$. Here, for all $\psi \in Pred$ and $s \in \Sigma$,

$$\begin{array}{ll} & s \in \mathsf{wrp}_\Omega.\pi.\psi \\ \Longleftrightarrow & \quad \{\text{Definition of wrp above}\} \\ & \forall\sigma : (s, \sigma) \in D(\pi) : \sigma \in \psi \cup \Omega \\ \Longleftrightarrow & \quad \{\text{Trading the range}\} \\ & \forall\sigma : \sigma \in \neg\psi : (s, \sigma) \notin D(\pi) \\ \Longleftrightarrow & \quad \{\text{Definition of } D(\pi)\} \\ & \forall s' : s' \in \neg\psi : s \in \mathsf{xwrp}.\Omega.\pi.\neg s' \end{array}$$

[2] Here we benefit from our set-theoretic concept of predicates. There are models of quite more abstract predicate concepts where it is not possible to select single states satisfying a given predicate. Consider, for instance, the abstract relation algebra [74, 78] which has models without so-called *points* [52, 74]. However, by set-theoretic axioms we are enabled to select predicates like $\Sigma \setminus \{s'\}$. Furthermore, and even more interesting, all oncoming calculations remain valid for *atomistic* predicate concepts where so-called *point-predicates*, $(d.s)$, defined by

$$(d.s).t \iff s = t$$

are guaranteed to exist for all states s. Predicate $\Sigma \setminus \{s'\}$ corresponds to $\neg(d.s')$ in this case.

[3] Note that the generalized pairing condition (Lemma 6.1.1) remains valid for this particular wrp-transformer as no restrictions concerning the underlying relational semantics are relevant.

$$
\begin{array}{ll}
\Longleftrightarrow & \{\text{Definition of greatest lower bound}\} \\
& s \in \bigcap_{s' \in \neg\psi} \mathsf{xwrp}.\Omega.\pi.\neg s' \\
\Longleftrightarrow & \{\mathsf{xwrp}.\Omega.\pi \text{ is universally conjunctive, [AX2]}\} \\
& s \in \mathsf{xwrp}.\Omega.\pi.(\bigcap_{s' \in \neg\psi} \neg s') \\
\Longleftrightarrow & \{\text{See below}\} \\
& s \in \mathsf{xwrp}.\Omega.\pi.\psi \ ,
\end{array}
$$

where in the last step we observe that, for all $t \in \Sigma$,

$$
\begin{array}{ll}
& t \in \bigcap_{s' \in \neg\psi} \neg s' \\
\Longleftrightarrow & \{\text{Definition of greatest lower bound}\} \\
& \forall s' :\ s' \in \neg\psi :\ t \in \neg s' \\
\Longleftrightarrow & \{\text{Trading the range, } s', t \in \Sigma\} \\
& \forall s' :\ s' = t :\ s' \in \psi \\
\Longleftrightarrow & \{\text{One point rule}\} \\
& t \in \psi \ .
\end{array}
$$

This proves the claim for the special case $A = \Omega$. Now consider an arbitrary $A \neq \Omega$. Similarly we observe that, for all $s \in \Sigma$,

$$
\begin{array}{ll}
& s \in \mathsf{wrp}_A.\pi.\mathtt{true} \\
\Longleftrightarrow & \{\text{Definition of } \mathsf{wrp} \text{ above}\} \\
& \forall \sigma :\ (s,\sigma) \in D(\pi) :\ \sigma \in \mathtt{true} \cup A \\
\Longleftrightarrow & \{\text{Trading the range}\} \\
& \forall \sigma :\ \sigma \in \Omega \setminus A :\ (s,\sigma) \notin D(\pi) \\
\Longleftrightarrow & \{\text{Naming convention and definition of } D(\pi)\} \\
& \forall \omega :\ \omega \in \Omega \setminus A :\ s \in \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true} \\
\Longleftrightarrow & \{\text{Definition of greatest lower bound}\} \\
& s \in \bigcap_{\omega \in \Omega \setminus A} \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true} \\
\Longleftrightarrow & \{\mathsf{xwrp}.\bullet.\pi.\mathtt{true} \text{ is universally conjunctive, [AX2]}\} \\
& s \in \mathsf{xwrp}.(\bigcap_{\omega \in \Omega \setminus A} (\Omega \setminus \omega)).\pi.\mathtt{true} \\
\Longleftrightarrow & \{\text{Again, see below}\} \\
& s \in \mathsf{xwrp}.A.\pi.\mathtt{true} \ ,
\end{array}
$$

where completely analogously we have, for all $\omega' \in \Omega$,

$$
\begin{array}{ll}
& \omega' \in \bigcap_{\omega \in \Omega \setminus A} (\Omega \setminus \omega) \\
\Longleftrightarrow & \{\text{Definition of greatest lower bound}\}
\end{array}
$$

$$
\begin{array}{ll}
& \forall \omega : \ \omega \in \Omega \setminus A : \ \omega' \in (\Omega \setminus \omega) \\
\Longleftrightarrow & \quad \{\text{Complementation w.r.t. } \Omega \text{ and } \omega, \omega' \in \Omega\} \\
& \forall \omega : \ \omega \notin A : \ \omega' \neq \omega \\
\Longleftrightarrow & \quad \{\text{Trading the range}\} \\
& \forall \omega : \ \omega' = \omega : \ \omega \in A \\
\Longleftrightarrow & \quad \{\text{One point rule}\} \\
& \omega' \in A \ .
\end{array}
$$

From here it is easy to show that, for all $\psi \in \mathit{Pred}$ and $s \in \Sigma$,

$$
\begin{array}{ll}
& s \in \mathsf{wrp}_A.\pi.\psi \\
\Longleftrightarrow & \quad \{\text{Generalized pairing condition for } \mathsf{wrp}, \text{ i.e. Lemma 6.1.1}\} \\
& s \in \mathsf{wrp}_\Omega.\pi.\psi \wedge s \in \mathsf{wrp}_A.\pi.\mathtt{true} \\
\Longleftrightarrow & \quad \{\text{Results above}\} \\
& s \in \mathsf{xwrp}_\Omega.\pi.\psi \wedge s \in \mathsf{xwrp}_A.\pi.\mathtt{true} \\
\Longleftrightarrow & \quad \{\text{Generalized Pairing condition for } \mathsf{xwrp}, \text{ i.e. [AX1]}\} \\
& s \in \mathsf{xwrp}_A.\pi.\psi \ ,
\end{array}
$$

which proves the general case.
□

As shown in Sect. 6.1, the law of the excluded miracle for wrp is satisfied if and only if the underlying relational semantics is total. A corresponding result holds in the axiomatic world as shown below.

Lemma 6.2.3 (Totality of $D(\pi)$). The derived relation $D(\pi)$ is total if and only if the axiomatic law of the excluded miracle, i.e. [Excluded Miracle], holds for xwrp.

Proof.

$$
\begin{array}{ll}
& D(\pi) \text{ is total} \\
\Longleftrightarrow & \quad \{\text{Definition of a total relation (2.2)}\} \\
& \forall s :: \ (\exists \sigma :: \ (s, \sigma) \in D(\pi)) \\
\Longleftrightarrow & \quad \{\text{Definition of } D(\pi) \text{ and naming conventions}\} \\
& \forall s :: \ (\exists s' :: \ s \in \neg \mathsf{xwrp}.\Omega.\pi.\neg s') \ \vee \\
& \qquad (\exists \omega :: \ s \in \neg \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true}) \\
\Longleftrightarrow & \quad \{\text{Trading the range and negation}\} \\
& \forall s : \ (\forall s' :: \ s \in \mathsf{xwrp}.\Omega.\pi.\neg s') \ \wedge \\
& \qquad (\forall \omega :: \ s \in \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true}) : \ s \in \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Definition of greatest lower bound}\} \\
& \forall s : \ s \in \bigcap_{s'} \mathsf{xwrp}.\Omega.\pi.\neg s' \ \wedge \\
& \qquad s \in \bigcap_{\omega} \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true} : \ s \in \mathtt{false} \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\}
\end{array}
$$

$$\bigcap_{s'} \mathsf{xwrp}.\Omega.\pi.\neg s' \cap \bigcap_{\omega} \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true} \subseteq \mathtt{false}$$

$\Longleftrightarrow$ {Universal Conjuctivity, axiom [AX2]}

$$\mathsf{xwrp}.\Omega.\pi.(\bigcap_{s'} \neg s') \cap \mathsf{xwrp}.(\bigcap_{\omega}(\Omega \setminus \omega)).\pi.\mathtt{true} \subseteq \mathtt{false}$$

$\Longleftrightarrow$ {$\bigcap_{s'} \neg s' = \mathtt{false}$ and $\bigcap_{\omega}(\Omega \setminus \omega) = \emptyset$, have a look at the proof of Lemma 6.2.2}

$$\mathsf{xwrp}.\Omega.\pi.\mathtt{false} \cap \mathsf{xwrp}.\emptyset.\pi.\mathtt{true} \subseteq \mathtt{false}$$

$\Longleftrightarrow$ {Generalized pairing condition for xwrp, i.e. [AX1]}

$$\mathsf{xwrp}.\emptyset.\pi.\mathtt{false} \subseteq \mathtt{false}$$

□

So much for total relations. The last section was also concerned with univalent relations and indeed we can prove an analogue to Lemma 6.1.5, 2. Of course 1 remains valid for xwrp, too. This is not surprising because again we can use the axiomatic law of the excluded miracle, i.e. [Excluded Miracle] in this case, as a dummy for the derived relation $D(\pi)$ being total, cf. Lemma 6.2.3 and the proof of Lemma 6.1.5, 1.

Lemma 6.2.4 (Univalence of $D(\pi)$). The derived relation $D(\pi)$ is univalent if and only if π is axiomatically univalent, i.e. if property [Univalence] holds for π.

Proof.

$D(\pi)$ is univalent

$\Longleftrightarrow$ {First three steps in the proof of Lemma 6.1.5, 2}

$$\forall s, \sigma, \sigma', A, \psi : \ (s,\sigma) \in D(\pi) \wedge (s,\sigma') \in D(\pi) : \\ (\sigma \in \neg\psi \cup \Omega \setminus A) \vee (\sigma' \in \psi \cup A)$$

$\Longleftrightarrow$ {Trading the range, calculus}

$$\forall s, \sigma, \sigma', A, \psi : \ (\sigma \in \psi \cup A) \wedge (\sigma' \in \neg\psi \cup \Omega \setminus A) : \\ (s,\sigma) \notin D(\pi) \vee (s,\sigma') \notin D(\pi)$$

$\Longleftrightarrow$ {Unnesting, $\forall$ distributes over $\vee$, trade the range and rename the dummy σ'}

$$\forall s, A, \psi :: \ (\forall \sigma : \ \sigma \in \psi \cup A : \ (s,\sigma) \notin D(\pi)) \ \vee \\ (\forall \sigma : \ \sigma \in \neg\psi \cup \Omega \setminus A : \ (s,\sigma) \notin D(\pi))$$

$\Longleftrightarrow$ {Splitting the range}

$$\forall s, A, \psi :: \ ((\forall \sigma : \ \sigma \in \psi : \ (s,\sigma) \notin D(\pi)) \wedge \\ (\forall \sigma : \ \sigma \in A : \ (s,\sigma) \notin D(\pi))) \ \vee \\ ((\forall \sigma : \ \sigma \in \neg\psi : \ (s,\sigma) \notin D(\pi)) \wedge \\ (\forall \sigma : \ \sigma \in \Omega \setminus A : \ (s,\sigma) \notin D(\pi)))$$

$\Longleftrightarrow$ {Naming conventions and definition of $D(\pi)$}

$$\forall s, A, \psi :: \ ((\forall s' : \ s' \in \psi : \ s \in \mathsf{xwrp}.\Omega.\pi.\neg s') \wedge$$

$$
\begin{array}{ll}
 & (\forall \omega :\ \omega \in A :\ s \in \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true}))\ \vee \\
 & ((\forall s' :\ s' \in \neg\psi :\ s \in \mathsf{xwrp}.\Omega.\pi.\neg s') \wedge \\
 & (\forall \omega :\ \omega \in \Omega \setminus A :\ s \in \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true})) \\
\Longleftrightarrow & \quad \{\text{Definition of greatest lower bound}\} \\
 & \forall s, A, \psi ::\ (s \in \bigcap_{s' \in \Psi} \mathsf{xwrp}.\Omega.\pi.\neg s'\ \wedge \\
 & \quad s \in \bigcap_{\omega \in A} \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true})\ \vee \\
 & \quad (s \in \bigcap_{s' \in \neg\Psi} \mathsf{xwrp}.\Omega.\pi.\neg s'\ \wedge \\
 & \quad s \in \bigcap_{\omega \in \Omega \setminus A} \mathsf{xwrp}.(\Omega \setminus \omega).\pi.\mathtt{true}) \\
\Longleftrightarrow & \quad \{\text{Universal conjunctivity ([AX2]) and calculations} \\
 & \quad \text{similar to the proof of Lemma 6.2.2}\} \\
 & \forall s, A, \psi ::\ (s \in \mathsf{xwrp}.\Omega.\pi.\neg\psi \wedge s \in \mathsf{xwrp}.(\Omega \setminus A).\pi.\mathtt{true})\ \vee \\
 & \quad (s \in \mathsf{xwrp}.\Omega.\pi.\psi \wedge s \in \mathsf{xwrp}.A.\pi.\mathtt{true}) \\
\Longleftrightarrow & \quad \{\text{Generalized pairing condition for } \mathsf{xwrp}\text{, i.e. [AX1]}\} \\
 & \forall s, A, \psi ::\ s \in \mathsf{xwrp}.(\Omega \setminus A).\pi.\neg\psi\ \vee\ s \in \mathsf{xwrp}.A.\pi.\psi \\
\Longleftrightarrow & \quad \{\text{Trading the range}\} \\
 & \forall s, A, \psi :\ s \in \neg\mathsf{xwrp}.(\Omega \setminus A).\pi.\neg\psi :\ s \in \mathsf{xwrp}.A.\pi.\psi \\
\Longleftrightarrow & \quad \{\text{Set inclusion}\} \\
 & \forall A, \psi ::\ \neg\mathsf{xwrp}.(\Omega \setminus A).\pi.\neg\psi \subseteq \mathsf{xwrp}.A.\pi.\psi
\end{array}
$$

□

For the sake of completeness we present the missing combination of Lemmas 6.2.3 and 6.2.4.

Lemma 6.2.5 ($D(\pi)$ and determinism). The derived relation $D(\pi)$ is a function, i.e. π is deterministic in the relational sense of Sect. 2.1, if and only if π is axiomatically deterministic, i.e. if property [Determinism] holds for π.
□

This lemma finishes our considerations in the current abstract scenario so let us resume the main issues. For a given program π it is possible to derive a (total, univalent resp. functional) relational semantics $D(\pi)$ from an axiomatically defined family of transformer $\mathsf{xwrp}.\bullet.\pi$ satisfying axioms [AX1] and [AX2] (and additionally the axiomatic law of the excluded miracle [Excluded Miracle], law [Univalence] resp. law [Determinism]) which carries the same information because otherwise lost information could be retrieved in miraculous ways. This was shown by establishing that, for all $A \subseteq \Omega$, the derived predicate transformer $\mathsf{wrp}_A.\pi$ equals the initially given $\mathsf{xwrp}.A.\pi$, and the results in parentheses were shown directly. In this sense we solved our exercise what expressiveness concerns and we

like to close this section with the following little summary which takes the initial observation, Lemma 6.2.1, into account and which is pictured in Fig. 6.1.

Theorem 6.2.1. Under the assumptions [AX1] and [AX2] (and [Excluded Miracle], [Univalence] resp. [Determinism]) a relativized predicate transformer semantics is as expressive as a (total, univalent resp. functional) relational semantics. □

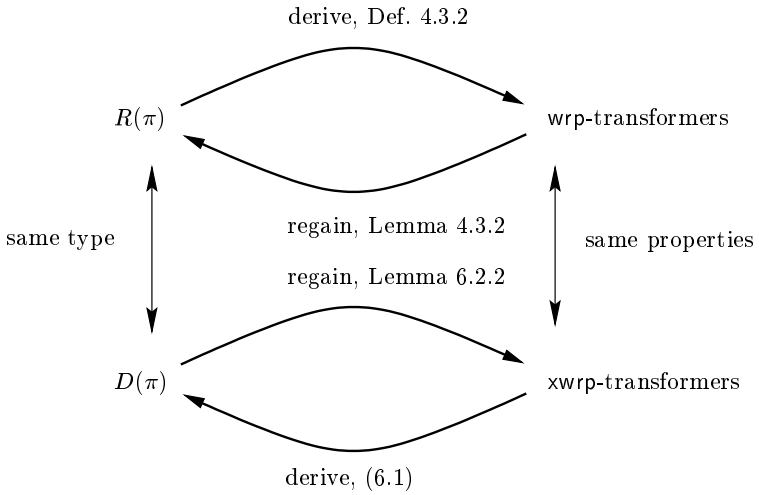

Fig. 6.1. On the expressiveness: Relational semantics vs. wrp-semantics.

7. Exemplary wrp-Semantics

Up to the present, the wrp-family was – quite abstractly – introduced in order to facilitate reasoning about the control-flow aspects of imperative, strictly transformational and realistic programs and consequently to support practical translation verification. Now it is time to become a little more demonstrative and in particular to embrace the title of the present thesis. Therefore, the present chapter is concerned with the very basic concepts of programming languages which cover most of the customary ones. More specifically, concrete wrp-semantics for the exemplarily chosen languages will be derived to show what they look like and to substantiate oncoming discussions concerning optimizations and translation verification.

In fact, two extreme instances of programming languages are discussed. Firstly, and this class of languages is typically heavily neglected what comprehensible semantics concerns, a machine language, to be precise an assembly language, is considered. In detail, the presented (abstract) machine language consists of labels, assignments, conditional jumps and subroutine-calls and -returns. It is intended to capture the essence of flat, unstructured assembly code[1] because we refrain from going down to the basics of concrete machine code as there are, e.g., load and store instructions on the byte-or-whatever-level and jump-destinations expressed by means of code-lengths. Instead, the mentioned instructions are supposed to be "macros" which can be refined by (sequences of) actual machine instructions in a later stage of the implementation in the real world. In this sense, the assembly language presented here might be seen as a stepping stone on the way down to actual binary machine code. The ultimative reference will be an operational semantics as it is common for assembly languages. Since the language is flat, i.e. without any inner structure, one cannot expect a nice and structured predicate transformer semantics but it is possible to derive sufficiently strong predicate transformer laws that support a semantically consistent reasoning about entire machine programs in an abstract fashion without executing the code operationally step-by-step. We like to mention that this assembly language was firstly presented in [63] where it plays the role of a target language of a "proved correct" translation, see Sect. 8.2 of the present thesis for more details.

Secondly, and more traditionally in the field of program verification, there is the class of WHILE-languages, the programs of which typically consist of assignments, sequential compositions, conditionals, loops and – generally not considered in a more theoretical setup like the present – procedure declarations and calls. We

[1] The fact that flat, unstructured assembly code is considered is not that self-evident as it seems. Sometimes structured assembly-languages are discussed, e.g. in [65], in order to ease translation correctness proofs, see also the comments in Sect. 8.3.

therefore introduce a language consisting of the constructs just mentioned which is taken as a very prototypic and general instance of the class of WHILE-languages covering most of the features they have in common. The language discussed here is in a sense more exciting as its procedure-concept allows nesting of procedures without naming restrictions (blockwise re-declarations). Again, the semantics will be given operationally and the major effort will be to derive a strictly compositional predicate transformer semantics using environments that map procedure identifiers to meanings of their bodies. It is easy to derive a predicate transformer semantics using dictionaries that map identifiers to programs – in fact, the operational semantics will use dictionaries – but, unfortunately, this semantics turns out to be not very compositional. It is a non-trivial exercise to show that both semantical representations coincide, the one that can be easily derived and the one which is more desirable.

In both languages we concentrate on the control-flow aspects of program execution. In particular, we have a rather superficial view on data. Both languages are assumed to work on a state space with named variables and we provide instructions embodying entire (Boolean) expressions, the semantics of which is assumed to be given by abstract evaluation functions. This allows to forget about the actual implementation of the evaluation of (Boolean) expressions which, after all, heavily depends on the concrete model resp. machine in question. It is noteworthy that both languages are allowed to produce (finite) errors. Typically, when reasoning about programs in more abstract ways, the underlying semantics is assumed to be total in the sense that execution of a command delivers a regular result (if it does terminate at all). The relative-correctness notion, however, is intended to get along with finite errors so it is appropriate and proximate to permit finite errors to emerge. Otherwise, the wrp-semantics would be rather senseless as the features which distinguish wrp from wp or wlp could barely be exploited.

The ultimate reference for both languages is an operational semantics which is the most common way of describing the behavior of programs and which promises to let errors due to misunderstandings become rare. But since reasoning in terms of a fine-grained operational semantics is quite error-prone and clumsy a more abstract view on program execution – here: wrp-transformers – is desirable. In the actual derivation of this wrp-semantics it turns out that it is helpful to have some meta-theorems available that relate entire computations on the operational level to predicate transformers. Thus, before going into the details of each of the languages, their shape of configurations and whatever, we start with an analysis of the very basics of operational semantics in general and show how to derive a fixpoint-oriented wrp-semantics from an operational semantics. As recursion is present in both languages we benefit from this investigation twice and in two flavors. The first is a concrete one: We directly obtain a fixpoint-characterization of the wrp-semantics for the machine language which is the key to the aspired laws, and also the task to derive a strictly compositional semantics for the source language relies on this presentation in large amounts. The second is of more general nature and of independent interest: The following intermezzo on operational semantics in general is hoped to procure a feeling for "adequate fixpoints". In particular it

suggests a "rule-of-thumb" saying that – in the area of weakest preconditions – greatest fixpoints resp. least fixpoints are the ones to choose whenever divergence is tolerated or not, respectively.

7.1 Fixpoint-Characterization of wrp

Let us set the stage for the more general considerations this section is about. Typically, an operational semantics is given by means of a *transition relation*, say '$\rightarrow$', on what is known under the name of *configurations*. The shape of configurations depends on the programming language in question, so let us investigate what they usually have in common. A characterizing property of a configuration is the presence of a *state component* which typically keeps the current values of the involved variables or memory-cells (i.e. a member of Σ in our setting) but which might also be an arbitrary complex entry dealing solely with data. Beside this, a configuration keeps the information concerning the control flow aspects of the present program and the current position. This component might solely consist of the program itself but there might be a variety of further components which are needed in order to accurately model the program running on a specific machine. Typical examples are stacks (for storing return addresses or local variables) or more abstract entries (like environments or dictionaries that map identifiers to values of some appropriate type). Even the entire memory of which the considered program is a part might be of interest. For our purposes, however, it turns out to be irrelevant what kind of specific information is kept in this second component so let us just suppose that it exists (a member of a set, say, C_1). Thus, a configuration is assumed to consist of a state component and a *control flow component* whatever the latter might precisely look like.

We range over the set of configurations $C \stackrel{\text{def}}{=} C_1 \times \Sigma$ by (c, s) where $c \in C_1$ represents the control flow component and $s \in \Sigma$ the state component. In our general setting, roughly speaking, the transition relation '$\rightarrow$' relates input configurations (c, s), the control flow component c is to be executed starting in state s, to output configurations (c', s'), execution terminated regularly in state s' leaving configuration c' to be executed next, or to final regular states s', the control flow component left nothing to do, or even to irregular outcomes ω because a run-time error occurred which forced the control flow component to stop propagating the obtained error message. Hence, in our vocabulary, we have

$$\rightarrow \ \subseteq \ C \times (C \cup \Sigma \cup \Omega) \ ,$$

and we explicitly allow a program to diverge spontaneously, i.e. $(c, s) \rightarrow \infty$, because this is rather natural in the following sense. We are not interested in the details of program execution – actually we do not even have programs – and in particular not in expression evaluation. Thus, in the sequel we will assume the existence of evaluation functions which allow to forget about the ways the results are obtained. However, if one is to implement these evaluation functions in some ways one might make use of loops and procedures which, after all, may diverge.

The considered operational semantics, given by '$\to$', can also be kept for a graph, a so-called *configuration graph*, with vertices of type C, Σ and Ω and edges given by the elements of '$\to$'. Note that configuration graphs are in no way restricted, typically they are cyclic and infinite. Computations on configurations correspond to paths in the configuration graph so the next task is to define a relation which lets us reason about finite *and* infinite paths. It is well known that the transitive closure of '$\to$' is the *least* relation '$\overset{+}{\to}$' which satisfies the tail recursive definition[2]

$$(c,s) \overset{+}{\to} x \iff^{\text{def}} (c,s) \to x \ \vee\ \exists c', s' :\ (c,s) \to (c',s') :\ (c',s') \overset{+}{\to} x \ , \quad (7.1)$$

where x is supposed to range over $C \cup \Sigma \cup \Omega$. With this definition, $(c,s) \overset{+}{\to} (c',s')$, $(c,s) \overset{+}{\to} s'$ and $(c,s) \overset{+}{\to} \omega$ represents a *finite* (and non-empty) path from (c,s) to (c',s'), s' and ω respectively. Infinite paths are disregarded because '$\overset{+}{\to}$' is defined to be the least solution of (7.1) so a solving relation containing both, finite and infinite paths, can only be of greater or equal size. Furthermore, the transitive closure of '$\to$' is defined to be the least transitive relation containing '$\to$'. However, infinite paths are obviously of interest, too, and we decide to capture them as follows. Based on the observation that a set of vertices, say S, describes an infinite path if each vertex in S has a successor in S we define the *successor set* S to be the *greatest* set satisfying[3]

$$S \overset{\text{def}}{=} \{(c,s) \mid \exists c', s' ::\ (c,s) \to (c',s') \ \wedge\ (c',s') \in S\} \ . \quad (7.2)$$

Here it is more obvious why the largest set is adequate; we like to collect all configurations which are part of some infinite path and not just of one particular. Furthermore $S = \emptyset$ is a trivial but little informative solution of (7.2). Combining (7.1) and (7.2) gives rise to the following definition: For configurations (c,s) and $x \in C \cup \Sigma \cup \Omega$:

$$(c,s) \rightsquigarrow x \iff^{\text{def}} (c,s) \overset{+}{\to} x \ \vee\ ((c,s) \in S \wedge x = \infty) \ . \quad (7.3)$$

Relation '$\rightsquigarrow$' captures both, finite and infinite paths; finite paths are obtained from '$\overset{+}{\to}$' and infinite paths are such which start in a configuration contained in S, the latter is made explicit by letting them "reach" ∞. (Note that relation '$\rightsquigarrow$' is generally not univalent, i.e. more than one outcome might be possible for a given initial state.)

A relation in the shape of '$\rightsquigarrow$' is precisely what we strive for, it capturing both, finite and infinite paths. Furthermore it is an easy exercise to show that '$\rightsquigarrow$' is also a solution of (7.1). Since this is an important and, moreover, a very useful observation it deserves an own quotable number.

Lemma 7.1.1 (Make a step).

$$(c,s) \rightsquigarrow x \iff (c,s) \to x \ \vee\ \exists c', s' :\ (c,s) \to (c',s') :\ (c',s') \rightsquigarrow x$$

[2] As '$\to$' is a relation one can also express the transitive closure of '$\to$' by $\mu X(\to \cup \to X)$ where sequential composition, denoted by juxtaposition here, of inhomogeneous relations – like '$\to$' – is defined as done in Sect. 5.1.

[3] Similarly, the set of all vertices lying on an infinite path corresponds to the relational vector $\nu X(\to X)$.

Proof. The details are left to the reader, just unroll the definitions and shunt the parts accordingly.
□

As we have seen, the weakest solution of (7.1) yields the transitive closure of '$\rightarrow$' which disregards infinite paths. From a semantical point of view this definition is *angelic* because the possibility of divergence is neglected in the sense that whenever an initial state might give rise to a diverging computation one does not care about this particular outcome and focuses on finitely reachable outcomes which, seen from this perspective, are actually of more interest (this corresponds to partial correctness). Let us therefore have a look at the *greatest* solution of (7.1). Obviously infinite paths are contained but the problem we are faced with is the following. Suppose there exists an infinite path starting in (c, s) such that $(c, s) \rightsquigarrow \infty$. In particular this infinite path can be used to show that

$$\exists c', s' : \ (c, s) \rightarrow (c', s') : \ (c', s') \rightsquigarrow \sigma \ \ ,$$

and this for all for $\sigma \in \Sigma \cup \Omega$! In the very end, $(c, s) \rightsquigarrow \sigma$ holds for all σ but this is not what we like to model. Thus, roughly and relationally speaking, an infinite path overwrites all information concerning finite paths. Whenever a diverging run may occur all possible terminating runs are disregarded. From a semantical point of view this is a *demonic* definition because divergence is treated as the worst case that can occur; whenever a computation can diverge it will. In contrary to the angelic point of view a diverging run will start whenever this is possible and all finite computations are neglected (this corresponds to total correctness). The moral of the tale is that '$\rightsquigarrow$' – the relation which is of actual interest – is neither the least nor the greatest solution of (7.1) so it seems necessary to define '$\rightsquigarrow$' the way it was done. This is not the time and place to foreclose the freedom to talk about variations of partial *and* total correctness, in this sense (7.3) is an *erratic* definition.

Let us now come to what this section is devoted to: A fixpoint characterization of wrp. The first step towards this goal is to define the wrp-transformers in this slightly different scenario where a directly underlying relational semantics is not at hand. One could, however, derive a relational semantics from the operational by defining

$$R(c) = \{(s, \sigma) \mid (c, s) \rightsquigarrow \sigma\}$$

first but we refrain from this notational detour. Instead we define the wrp-transformers directly – but equivalently – as follows: For a set $A \subseteq \Omega$ of irregular outcomes to be accepted, a control-flow component $c \in C_1$, a predicate $\psi \in \mathit{Pred}$ and a state s:[4]

[4] This presentation is mostly due to "historic" reasons but it also has some more notational and calculational advantages. One can equivalently and customary define

$$s \in \mathsf{wrp}_A.c.\psi \ \stackrel{\text{def}}{\Longleftrightarrow} \ \forall \sigma : \ (c, s) \rightsquigarrow \sigma : \ \sigma \in \psi \cup A \ \ ,$$

in particular we specify

$$\neg((c, s) \rightsquigarrow \Omega \setminus A) \ \stackrel{\text{def}}{\Longleftrightarrow} \ \forall \omega : \ (c, s) \rightsquigarrow \omega : \ \omega \in A \ \ .$$

$$s \in \mathsf{wrp}_A.c.\psi \overset{\text{def}}{\iff} \neg((c,s) \rightsquigarrow \Omega \setminus A) \wedge (\forall s' :\ (c,s) \rightsquigarrow s' :\ s' \in \psi)\ .$$

Again, a state satisfies the weakest relative precondition of c w.r.t. ψ and A if no path starting in (c,s) ends in a vertex representing an erroneous outcome contained in $\Omega \setminus A$ (note that, by definition (7.3), paths representing diverging computations "reach" ∞) and if all regular states which a reachable via a path starting in (c,s) satisfy the postcondition ψ.

One of the basic properties of '$\rightsquigarrow$' and the graph-oriented view is the following. Consider a configuration (c,s) such that s satisfies $\mathsf{wrp}_A.c.\psi$, i.e. all paths starting in (c,s) lead to vertices satisfying the postcondition resp. to be accepted. Then we can decompose each of those paths into a first single step and a remaining path and we can be sure that each of these vertices reachable in one step enjoys this property again (for its own control-flow component and an accordingly modified state). The converse is also valid and, more formally, this property is formulated as follows.

Lemma 7.1.2 (Unroll wrp).

$$\begin{aligned}
&s \in \mathsf{wrp}_A.c.\psi \iff \\
&\quad \neg((c,s) \to \Omega \setminus A) \wedge \\
&\quad (\forall s' :\ (c,s) \to s' :\ s' \in \psi) \wedge \\
&\quad (\forall c', s' :\ (c,s) \to (c',s') :\ s' \in \mathsf{wrp}_A.c'.\psi)
\end{aligned}$$

Proof.

$$\begin{aligned}
& s \in \mathsf{wrp}_A.c.\psi \\
\iff & \quad \{\text{Definition of } \mathsf{wrp}\} \\
& \neg((c,s) \rightsquigarrow \Omega \setminus A)\ \wedge\ (\forall s' :\ (c,s) \rightsquigarrow s' :\ s' \in \psi) \\
\iff & \quad \{\text{Make a step (Lemma 7.1.1) and distribute}\} \\
& \neg((c,s) \to \Omega \setminus A) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') :\ \neg((c',s') \rightsquigarrow \Omega \setminus A)) \wedge \\
& (\forall s'' :\ ((c,s) \to s'')\ \vee\ (\exists c', s' :\ (c,s) \to (c',s') :\ (c',s') \rightsquigarrow s'') : \\
& \qquad s'' \in \psi) \\
\iff & \quad \{\text{Splitting the range and } \exists\text{-introduction}\} \\
& \neg((c,s) \to \Omega \setminus A) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') :\ \neg((c',s') \rightsquigarrow \Omega \setminus A)) \wedge \\
& (\forall s'' :\ (c,s) \to s'' :\ s'' \in \psi) \wedge \\
& (\forall c', s', s'' :\ (c,s) \to (c',s') \wedge (c',s') \rightsquigarrow s'' :\ s'' \in \psi) \\
\iff & \quad \{\text{Unnesting}\} \\
& \neg((c,s) \to \Omega \setminus A) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') :\ \neg((c',s') \rightsquigarrow \Omega \setminus A)) \wedge \\
& (\forall s'' :\ (c,s) \to s'' :\ s'' \in \psi) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') :\ (\forall s'' :\ (c',s') \rightsquigarrow s'' :\ s'' \in \psi))
\end{aligned}$$

$$
\begin{array}{ll}
\Longleftrightarrow & \{\forall \text{ distributes over } \wedge\} \\
& \neg((c,s) \to \Omega \setminus A) \wedge \\
& (\forall s'' :\ (c,s) \to s'' :\ s'' \in \psi) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') : \\
& \qquad \neg((c',s') \rightsquigarrow \Omega \setminus A)\ \wedge\ (\forall s'' :\ (c',s') \rightsquigarrow s'' :\ s'' \in \psi)) \\
\Longleftrightarrow & \{\text{Definition of } \mathsf{wrp} \text{ and renaming a dummy}\} \\
& \neg((c,s) \to \Omega \setminus A) \wedge \\
& (\forall s' :\ (c,s) \to s' :\ s' \in \psi) \wedge \\
& (\forall c', s' :\ (c,s) \to (c',s') :\ s' \in \mathsf{wrp}_A.c'.\psi
\end{array}
$$

□

Lemma 7.1.2 is nice because it shows an iterated occurrence of wrp; this suggests to define a function in such ways that wrp becomes a fixpoint of it. First of all we make the technical observation that the wrp-transformer is of type

$$T \stackrel{\text{def}}{=} 2^{\Omega} \to (\mathcal{C}_1 \to PTrans)\ .$$

Then we define

$$D \in (T \to T)$$

which is, for a "transformer" $f \in T$, a set of outcomes $A \subseteq \Omega$, a control-flow component $c \in C_1$, a predicate $\psi \in Pred$ and states $s \in \Sigma$, defined as follows.

$$
\begin{array}{l}
s \in D.f.A.c.\psi \stackrel{\text{def}}{\Longleftrightarrow} \\
\quad \neg((c,s) \to \Omega \setminus A) \wedge \\
\quad (\forall s' :\ (c,s) \to s' :\ s' \in \psi) \wedge \\
\quad (\forall c', s' :\ (c,s) \to (c',s') :\ s' \in f.A.c'.\psi)
\end{array}
\tag{7.4}
$$

Now, in fact, it is easy to see that the transformer $\mathsf{wrp} \in T$ is a fixpoint of D, just replace f by wrp in D's definition (7.4) and apply Lemma 7.1.2 (note that $\mathsf{wrp}_A.c$ is a condensed denotation for $\mathsf{wrp}.A.c$). This observation is part of what we are looking for so it is worth a theorem.

Theorem 7.1.1 (wrp *is* a fixpoint of D).

$$D.\mathsf{wrp} \ =\ \mathsf{wrp}$$

□

The set $(T \to T)$ is a complete lattice and the function D is obviously monotonic. Hence, by the fixpoint theorem of Knaster and Tarski we could have concluded the existence of fixpoints without even knowing them explicitly instead of finding a solution in a constructive way. Furthermore, the fixpoint theorem states that D has a least and a greatest fixpoint and the question to be discussed now is which of the two extreme instances is adequate depending on the arguments. Operationally speaking, one has to distinguish the cases whether infinite paths are tolerated or not; spoken in terms of relative correctness one has to respect whether divergence is to be accepted or not, i.e $\infty \in A$ or $\infty \notin A$.

We start with a variation of partial correctness so assume $\infty \in A$. To prove $\mathsf{wrp}_A.c.\psi = \nu D.A.c.\psi$ we consider an arbitrary fixpoint f of D, i.e. $f \in T$ with $D.f = f$. We can show that f is less than wrp w.r.t. the order on T so the claim follows because wrp is a fixpoint of D which is greater than or equal to any other.[5] To show $f.A.c.\psi \subseteq \mathsf{wrp}_A.c.\psi$ we take an initial state $s \in f.A.c.\psi$. The claim is $s \in \mathsf{wrp}_A.c.\psi$, i.e. $\forall \sigma : \ (c,s) \rightsquigarrow \sigma : \ \sigma \in \psi \cup A$, and we thus choose an arbitrary σ with $(c,s) \rightsquigarrow \sigma$. Note that we are done if $\sigma = \infty$ because we accept divergence here, $\infty \in A$. Therefore, it suffices to consider finitely reachable outcomes only, and consequently the overall claim $s \in f.A.c.\psi \implies s \in \mathsf{wrp}_A.c.\psi$ can be shown by induction on finite paths in the following way. We like to prove

$$\forall n \in \mathbb{N}, c, s : \ s \in f.A.c.\psi : \ (\forall \sigma : \ (c,s) \rightarrow^n \sigma : \ \sigma \in \psi \cup A) \ , \tag{7.5}$$

by induction on n where $(c,s) \rightarrow^n \sigma$ denotes a finite path of length n from configuration (c,s) to outcome σ.
Base case: Take some arbitrary c, s and assume $s \in f.A.c.\psi$. As $D.f = f$ we have $s \in D.f.A.c.\psi$ and thus particularly

$$\begin{array}{ll} & \neg((c,s) \rightarrow \Omega \setminus A) \wedge (\forall s' : \ (c,s) \rightarrow s' : \ s' \in \psi) \\ \iff & \quad \{\text{Conventions and } \forall \text{ distributes over } \wedge\} \\ & \forall \sigma : \ (c,s) \rightarrow \sigma : \ \sigma \in \psi \cup A \ . \end{array}$$

Induction step: Assume the claim (7.5) to hold for all c, s and an arbitrary but fixed n. Now take some c and s and observe that

$$\begin{array}{ll} & s \in f.A.c.\psi \\ \implies & \quad \{D.f = f, \text{ definition of } D, \text{ forget about the first conjuncts}\} \\ & (\forall c', s' : \ (c,s) \rightarrow (c',s') : \ s' \in f.A.c'.\psi) \\ \implies & \quad \{\text{Induction hypothesis}\} \\ & \forall c', s' : \ (c,s) \rightarrow (c',s') : \ (\forall \sigma : \ (c',s') \rightarrow^n \sigma : \ \sigma \in \psi \cup A) \\ \iff & \quad \{\text{Combine the paths}\} \\ & \forall \sigma : \ (c,s) \rightarrow^{n+1} \sigma : \ \sigma \in \psi \cup A \ , \end{array}$$

which completes the proof of

Theorem 7.1.2 (Choose ν for $\infty \in A$). If $\infty \in A$ then

$$\mathsf{wrp}_A \ = \ \nu D.A \ .$$

□

The greatest fixpoint of D is the adequate one if we tolerate divergence, just as expected. By symmetry we conjecture the least one to be appropriate if divergence is to be rejected and this is proved as follows.

Again, consider another fixpoint f of D, i.e. $D.f = f$, and assume that $\infty \notin A$. This time we start to shuffle the components:

[5] The fixpoint-theorem of Knaster and Tarski particularly says that the set of fixpoints of a monotonic function on a complete lattice forms also a complete lattice so this conclusion makes sense.

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.c.\psi \subseteq f.A.c.\psi \\
\Longleftrightarrow & \quad \{\text{Set-inclusion and definition of } \mathsf{wrp}\} \\
 & \forall s:\ \neg((c,s) \rightsquigarrow \Omega \setminus A) \wedge (\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi):\ s \in f.A.c.\psi \\
\Longleftrightarrow & \quad \{\infty \notin A\} \\
 & \forall s:\ \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & \quad \neg((c,s) \rightsquigarrow \infty) \wedge \\
 & \quad (\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi): \\
 & \quad\ s \in f.A.c.\psi \\
\Longleftrightarrow & \quad \{\text{Trading the range twice}\} \\
 & \forall s:\ \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & \quad (\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi) \wedge \\
 & \quad s \notin f.A.c.\psi: \\
 & \quad\ (c,s) \rightsquigarrow \infty \\
\Longleftrightarrow & \quad \{\text{See definition below}\} \\
 & \forall s:\ P(c,s):\ (c,s) \rightsquigarrow \infty\ ,
\end{array}
$$

where we define a "path-predicate" $P(c,s)$ by

$$
\begin{array}{l}
P(c,s) \stackrel{\text{def}}{\Longleftrightarrow} \\
\neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
(\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi) \wedge \\
s \notin f.A.c.\psi\ .
\end{array}
$$

Vocalized, the goal can be achieved by constructing an infinite (or finite) path to ∞ starting in (c,s) under the assumption that $P(c,s)$ holds. This is shown as follows.

$$
\begin{array}{ll}
 & P(c,s) \\
\Longleftrightarrow & \quad \{\text{Definition of } P,\ D.f = f\} \\
 & \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & (\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi) \wedge \\
 & s \notin D.f.A.c.\psi \\
\Longleftrightarrow & \quad \{\text{Definition of } D\} \\
 & \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & (\forall s':\ (c,s) \rightsquigarrow s':\ s' \in \psi) \wedge \\
 & (\ ((c,s) \to \Omega \setminus A) \vee \\
 & \quad (\exists s':\ (c,s) \to s':\ s' \notin \psi) \vee \\
 & \quad (\exists c', s':\ (c,s) \to (c',s') : s' \notin f.A.c'.\psi)\ ) \\
\Longrightarrow & \quad \{\text{Distribute the first disjunct;} \\
 & \quad\ \text{if } \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \text{ and } ((c,s) \to \Omega \setminus A) \\
 & \quad\ \text{then } ((c,s) \to \infty) \text{ follows}\}
\end{array}
$$

$$
\begin{array}{ll}
 & ((c,s) \to \infty) \vee \\
 & (\ \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & \quad (\forall s' :\ (c,s) \rightsquigarrow s' :\ s' \in \psi) \wedge \\
 & \quad (\ (\exists s' :\ (c,s) \to s' :\ s' \notin \psi)\ \vee \\
 & \quad\quad (\exists c', s' :\ (c,s) \to (c',s') : s' \notin f.A.c'.\psi)\ )\ ) \\
\Longrightarrow & \quad \{\text{Distribute the last but one disjunct which} \\
 & \quad\quad \text{contradicts } (\forall s' :\ (c,s) \rightsquigarrow s' :\ s' \in \psi)\} \\
 & ((c,s) \to \infty) \vee \\
 & (\ \neg((c,s) \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & \quad (\forall s' :\ (c,s) \rightsquigarrow s' :\ s' \in \psi) \wedge \\
 & \quad (\exists c', s' :\ (c,s) \to (c',s') : s' \notin f.A.c'.\psi)\ ) \\
\Longrightarrow & \quad \{\text{“Make a step” (Lemma 7.1.1) and forget about some} \\
 & \quad\quad \text{conjuncts, then choose “best of both worlds”}\} \\
 & ((c,s) \to \infty) \vee \\
 & (\exists c', s' :\ (c,s) \to (c',s') :\ \neg((c',s') \rightsquigarrow \Omega \setminus (A \cup \{\infty\})) \wedge \\
 & \qquad\qquad\qquad\qquad (\forall s'' :\ (c',s') \rightsquigarrow s'' :\ s'' \in \psi) \wedge \\
 & \qquad\qquad\qquad\qquad s' \notin f.A.c'.\psi) \\
\Longleftrightarrow & \quad \{\text{Definition of } P\} \\
 & ((c,s) \to \infty) \vee \\
 & (\exists c', s' :\ (c,s) \to (c',s') :\ P(c',s'))
\end{array}
$$

Thus, assuming $P(c,s)$ to hold lets us conclude the existence of a (finite) one-step computation from (c,s) to ∞ which particularly implies $(c,s) \rightsquigarrow \infty$ or the existence of a successive configuration (c',s') satisfying P again. Following these successive configurations lets us construct a – finite or infinite – path to ∞ anyhow so $(c,s) \rightsquigarrow \infty$ holds as required.

Here it was shown that wrp is a fixpoint of D which is less than or equal to any fixpoint of D, consequently it is the least fixpoint.

Theorem 7.1.3 (Choose μ for $\infty \notin A$). If $\infty \notin A$ then

$$\mathsf{wrp}_A \ = \ \mu D.A \ .$$

□

This theorem closes the intermezzo on fixpoints in connection with weakest preconditions and we will benefit from these insights many a time, both directly and inspirationally. Though the results presented here cannot always be directly applied they should nevertheless be understood as a justification for the following rule-of-thumb suited to weakest preconditions.

“Choose the greatest fixpoint if $\infty \in A$ and the least otherwise.”

7.2 Preparations and Notations

As mentioned before we focus on the control flow aspects of program execution. In particular, we refrain from going into details of (Boolean) expression evaluation and the treatment of data in general. However, since realistic programming languages obviously handle with data – this is what they are basically intended to do – the present section is devoted to a brief introduction to our more abstract view on data, to the ways we assume evaluations of (Boolean) expressions to be performed and also to some helpful notational sugar.

For the remainder we suppose given three additional sets of syntactic objects: a set *Var* of *variables* x, a set *Expr* of *expressions* e, and a set *BExpr* of *Boolean expressions* b. We do not care about the concrete structure, i.e. the syntax, of the latter two because this entails going into details of their evaluation what we want to avoid. Instead, we assume given two abstract evaluation functions,

$$\mathcal{E}(e) \in (\Sigma \to (\mathit{Val} \cup \Omega)) \quad \text{and} \quad \mathcal{B}(b) \in (\Sigma \to (\mathbb{B} \cup \Omega)) \ ,$$

which act as oracles and yield either the values of (Boolean) expressions or erroneous outcomes, and this in some – unknown and uninteresting – ways. Here, *Val* is supposed to be the value-set of variables, and $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ represents the truth-values. Intuitively, results $\mathcal{B}(b)(s) \in \Omega$ and $\mathcal{E}(e)(s) \in \Omega$ represent failures during the evaluation of (Boolean) expressions which are assumed to propagate on the state-level. Note that evaluations may even diverge. We already mentioned that this is reasonable in the sense that these evaluation functions may run forever if they are to be implemented in a later stage.

In the sequel, states are valuations of variables, i.e. $\Sigma = (\mathit{Var} \to \mathit{Val})$. As usual, $s\{x \mapsto v\}$ denotes a *variation*, i.e. an update of variable x's value in state s to v, more technically,

$$s\{x \mapsto v\}(y) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } x = y \\ s(y) & \text{if } x \neq y \end{cases} \ . \tag{7.6}$$

Obviously, a variation is a commutative operation for distinct variables, i.e.

$$s\{x \mapsto v\}\{y \mapsto w\} \ = \ s\{y \mapsto w\}\{x \mapsto v\}$$

if $x \neq y$. On the other side, $\psi[e/x]$, defined by

$$s \in \psi[e/x] \stackrel{\text{def}}{\iff} s\{x \mapsto \mathcal{E}(e)(s)\} \in \psi$$

denotes a *substitution* of e (better, e's value) for x in predicate ψ (notice that the substitution is only defined if $\mathcal{E}(e)(s) \in \mathit{Val}$). Due to the pointwise definition it is clear that a substitution is universally junctive, i.e. both universally disjunctive and universally conjunctive (in the predicate calculus setting of [18] the converse is also true: A function is a substitution if it is universally junctive, see [9]). Note that, in our setting, a substitution is a semantic operation on predicates rather than a syntactic operation on expressions. If, however, a "syntactical substitution" of expression e for variable x in expression f is of interest, e.g. denoted by $f\langle e/x\rangle$, its value can be defined by

$$\mathcal{E}(f\langle e/x\rangle)(s) \stackrel{\text{def}}{=} \mathcal{E}(f)(s\{x \mapsto \mathcal{E}(e)(s)\}) \ , \tag{7.7}$$

for states s with $\mathcal{E}(e)(s) \in \mathit{Val}$. A well-known law remains valid: Two successive semantical substitutions equal one semantical substitution of a syntactical substitution in the sense that

$$\psi[e/x][f/x] \;=\; \psi[e\langle f/x\rangle/x] \;. \tag{7.8}$$

In order to deal with partially defined expressions and for the sake of readability we assume special types of basic predicates. As evaluation of (Boolean) expressions may fail and because relative correctness is concerned with regular results and erroneous outcomes to be accepted the following predicates collect the corresponding states. They are, for (Boolean) expressions b and e and for $A \subseteq \Omega$, defined as follows:

$$\begin{aligned}
\mathsf{def}(e) &\stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in \mathit{Val}\} \;, \\
\mathsf{in}_A(e) &\stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in A\} \;, \\
\mathsf{def}(b) &\stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) \in \mathbb{B}\} \;, \text{ and} \\
\mathsf{in}_A(b) &\stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) \in A\} \;.
\end{aligned}$$

Their meaning should be intuitively clear: $\mathsf{def}(e)$ resp. $\mathsf{in}_A(e)$, for instance, is the set of states in which evaluation of e yields a regular result resp. an irregular outcome that is to be accepted. Semantical guards are based on the following two predicates which contain states satisfying a condition resp. not:

$$\begin{aligned}
(b = \mathsf{tt}) &\stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) = \mathsf{tt}\} \;, \text{ and} \\
(b = \mathsf{ff}) &\stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) = \mathsf{ff}\} \;.
\end{aligned}$$

Note that the evaluation of (Boolean) expressions can yield erroneous outcomes whereas the corresponding predicates cannot. By this the underlying logic of predicates we are reasoning in becomes total.

The predicates mentioned so far are concrete and due to their suggestive names more intuitive instances of two more general predicates. For a set of outcomes $O \subseteq \Sigma \cup \Omega$ and an expression e we define

$$\mathsf{in}_O(e) \;\stackrel{\text{def}}{=}\; \{s \mid \mathcal{E}(s) \in O\} \tag{7.9}$$

and analogously $\mathsf{in}_O(b)$ for Boolean expressions b.

These basic predicates give rise to some necessary or at least useful predicate transformers which semantically model some typical commands of imperative programming languages. By example let us consider an assignment, say "$x := e$". Operationally speaking, the semantics of an assignment is as follows. Firstly, the expression e is evaluated in some ways in some state s, in our scenario this evaluation is modeled by application of the evaluation function $\mathcal{E}$ to e and s. If evaluation delivers a regular result it is assigned to x, i.e. state s is varied accordingly. If evaluation of e fails and yields an erroneous outcome ω the acceptance of this failure depends on whether $\omega \in A$ or not. Hence, the predicate transformer modeling this assignment is given by

$$(x :=_A e).\psi \;\stackrel{\text{def}}{=}\; \mathsf{in}_A(e) \;\cup\; (\mathsf{def}(e) \cap \psi[e/x]) \;,$$

note that the guard $\mathsf{def}(e)$ guarantees that the substitution is well-defined. Another widely used command is the conditional, say guarded by b with branches P and Q where the latter are supposed to be weakest relative precondition predicate transformers. Operationally, the guard b is evaluated in some given state, and if evaluation terminates regularly yielding tt resp. ff the corresponding branch P resp. Q is chosen. Again, evaluation of b may also yield erroneous outcomes such that the semantical conditional we are interested in is given by

$$(P \lhd b/A \rhd Q).\psi \stackrel{\text{def}}{=} \mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap P.\psi) \cup ((b = \mathsf{ff}) \cap Q.\psi) \ .$$

The below law is well known from the refinement calculus: Sequential composition from the right *distributes over conditionals.*

Lemma 7.2.1 (Distribute conditional).

$$(P \lhd b/A \rhd Q) \,;\, R \ = \ (P \,;\, R) \lhd b/A \rhd (Q \,;\, R)$$

□

Yet another definition is appropriate here. The semantics of a while-loop can be nicely defined by means of a function $\mathcal{W}_{b,P} : PTrans \to PTrans$ which is defined by

$$\mathcal{W}_{b,P}(X) \stackrel{\text{def}}{=} (P \,;\, X) \lhd b/A \rhd Id \ , \tag{7.10}$$

where P is again supposed to be a weakest relative precondition transformer (and where A will be clear from the context). Operationally, a loop with body π is unrolled as long as the guard holds. Semantically spoken, in terms of predicate transformers, this reflects in taking an adequate – and we will see what this means in a while or may even obey the rule-of-thumb right now – fixpoint of $\mathcal{W}_{b,\mathsf{wrp}_A.\pi}$.

7.3 An Abstract Assembly Language

The language defined in this section is intended to capture the essence of flat and unstructured assembly code. In this, our main interest is a realistic treatment of control structures. Therefore, labels $l \in Lab$ are used to mark the destination of jump instructions as common in assembly languages. In order to keep things manageable, the language works on a state space with named variables and we provide instructions embodying entire (Boolean) expressions: $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$. Such instructions should be thought to be "macros" representing a sequence of more concrete assembly instructions; one can easily imagine that an assignment is a collection of load-, store- and arithmetic-operations on a real machine and likewise is a conditional jump. A language of this kind might be used as a stepping stone on the way down to actual binary machine code, and indeed the presented language here is *almost* an abstract view on an existing assembly language which is the Transputer-code in this case.[6] Moreover, the presented machine language is

[6] Though the assembly language considered here looks rather artificial, in particular its semantics, it can be kept for a more handy representation of real assembly code. The appendix is concerned with a justification that it is – in principle – possible to elaborate increasingly more abstract views on the Transputer such that finally a language and semantics similar to the one presented here can be derived.

slightly more realistic than common – what abstract views like ours typically concerns – because it is assumed to run on finite machines. In the present scenario this expresses in stack-overflow errors that may spontaneously arise whenever a return-address is pushed onto the return-stack. The possibility of resource-violation is modeled by non-determinism because otherwise the actual memory size and its current utilization must be taken into account which obviously would complicate the definitions, and relative correctness is intended to feature elegance.

7.3.1 Syntax

The set *Instr* consists of *instructions* of the following form. There is

– $\mathsf{asg}(x, e)$: an assignment instruction,
– $\mathsf{cj}(b, l)$: a conditional jump (on false) to label l,
– $\mathsf{jsr}(l)$: a subroutine jump to label l, and
– ret: a return jump.

We write $\mathsf{goto}(l)$ as a short hand notation for $\mathsf{cj}(\mathtt{false}, l)$. It represents an unconditional jump.

An *assembly (or machine) program* m is a finite sequence consisting of instructions and labels where we assume *unique labeling*. Concatenation of programs is denoted by an infix dot '$\cdot$' and m_i denotes the ith component, i.e. an instruction or a label, in the machine program $m = m_1 \cdot \ldots \cdot m_i \cdot \ldots \cdot m_n$. More formally, the set of assembly (or machine) programs is given by

$$MP \stackrel{\text{def}}{=} \{m \in (Instr \cup Lab)^* \mid \forall i, j : \ m_i = m_j \in Lab : \ i = j\} \ ,$$

and ε denotes the empty string, in particular ε is an assembly program. In a program m, the occurrences of labels inside cj and jsr instructions are called *applied*, the other occurrences are called *defining*; note that defining occurrences of labels are unique. A program m is called *closed* if every label that has an applied occurrence in m also has a defining occurrence. The set of *closed machine programs* is denoted by *CMP*, i.e.

$$CMP \stackrel{\text{def}}{=} \{m \in MP \mid m \text{ is closed}\} \ .$$

Here is an example of a closed program computing the factorial of x leaving the result in y.[7]

$$\mathsf{asg}(\mathsf{y}, 1) \cdot \mathsf{Loop} \cdot \mathsf{cj}(\mathsf{x} \neq 0, \mathsf{End}) \cdot \mathsf{asg}(\mathsf{y}, \mathsf{x} * \mathsf{y}) \cdot \mathsf{asg}(\mathsf{x}, \mathsf{x} - 1) \cdot \mathsf{goto}(\mathsf{Loop}) \cdot \mathsf{End}$$

[7] For the sake of comprehension we mention that this program is just a technically more manageable denotation of the program

```
        asg(y, 1)
Loop    cj(x ≠ 0, End)
        asg(y, x * y)
        asg(x, x − 1)
        goto(Loop)
End
```

which is a presentation each computer scientist should be familiar with and which exactly yields the one above if it is read line by line added by separating dots.

7.3.2 Basic operational semantics

A processor executing a machine program will typically use an instruction pointer that points to the next instruction to be executed at any given moment. For the reasoning about assembly code in a more algebraic fashion, however, it is more convenient to represent the current control point in a more symbolic manner: We partition the executed program m into two parts u, v such that $m = u \cdot v$ and that the next instruction to be executed is just the first instruction of v. Progress of execution can be nicely expressed by partitioning the same code sequence differently. The set PMP of *partitioned machine programs* contains the formal representations of programs together with an instruction pointer:

$$PMP \stackrel{\text{def}}{=} \{\langle u, v\rangle \mid u \cdot v \in CMP\} \ .$$

The partitioned program $\langle u, v\rangle \in PMP$, for example, where

$$u \ = \ \mathsf{asg}(\mathsf{y}, 1) \cdot \mathsf{Loop} \cdot \mathsf{cj}(\mathsf{x} \neq 0, \mathsf{End}) \cdot \mathsf{asg}(\mathsf{y}, \mathsf{x} * \mathsf{y}) \cdot \mathsf{asg}(\mathsf{x}, \mathsf{x} - 1)$$

and

$$v \ = \ \mathsf{goto}(\mathsf{Loop}) \cdot \mathsf{End}$$

represents the above factorial program just after an iteration of the body and just before the back-jump to the beginning of the loop.

Similarly, we prefer to work with a symbolic representation of the stack of return addresses; such a stack is necessary to execute jump-subroutine and return instructions. The idea is to use a stack of partitioned code sequences (modeled by a member of PMP^*) instead of a stack of (absolute) addresses. More precisely, at execution time each element of the PMP^*-stack represents the same program, i.e. the program in question, only the positions of the separating commas – marking the instructions to be executed just after the return – differ.

The basic semantics of the abstract assembly language is an operational semantics built around the ideas just described. It works on configurations of the form $\langle u, v, a, s\rangle$, where $\langle u, v\rangle \in PMP$ models the current control point ($u \cdot v$ is the present program and the first instruction of v is to be executed next), $a \in PMP^*$ is the symbolic representation of the return stack, and $s \in \Sigma$ is the current state. Thus,

$$\Gamma_{MP} \stackrel{\text{def}}{=} \{\langle u, v, a, s\rangle \mid \langle u, v\rangle \in PMP \wedge a \in PMP^* \wedge s \in \Sigma\}$$

is the set of *regular configurations* of the assembly language. In order to treat error situations, we use the members of Ω as *irregular configurations* and, of course, states $s \in \Sigma$ represent the results of regular terminating computations which could be called *final configurations*. Table 7.1 defines the transition relation $\rightarrow \subseteq \Gamma_{MP} \times (\Gamma_{MP} \cup \Sigma \cup \Omega)$ of an abstract machine executing assembly programs.

Let us consider the rules in more detail. [Asg1] is concerned with the execution of $\mathsf{asg}(x, e)$. If e evaluates without error to a value in the current state s the machine changes the value of x accordingly – the new state is $s\{x \mapsto \mathcal{E}(e)(s)\}$ – and transfers control to the subsequent instruction. This is nicely modeled by moving the $\mathsf{asg}(x, e)$ instruction from the start of the v component to the end of

[Asg1] $$\frac{\mathcal{E}(e)(s) \in \mathit{Val}}{\langle u, \mathsf{asg}(x,e) \cdot v, a, s\rangle \to \langle u \cdot \mathsf{asg}(x,e), v, a, s\{x \mapsto \mathcal{E}(e)(s)\}\rangle}$$

[Asg2] $$\frac{\mathcal{E}(e)(s) \in \Omega}{\langle u, \mathsf{asg}(x,e) \cdot v, a, s\rangle \to \mathcal{E}(e)(s)}$$

[Cj1] $$\frac{\mathcal{B}(b)(s) = \mathsf{tt}}{\langle u, \mathsf{cj}(b,l) \cdot v, a, s\rangle \to \langle u \cdot \mathsf{cj}(b,l), v, a, s\rangle}$$

[Cj2] $$\frac{\mathcal{B}(b)(s) = \mathsf{ff}\ ,\ \ u \cdot \mathsf{cj}(b,l) \cdot v = x \cdot l \cdot y}{\langle u, \mathsf{cj}(b,l) \cdot v, a, s\rangle \to \langle x, l \cdot y, a, s\rangle}$$

[Cj3] $$\frac{\mathcal{B}(b)(s) \in \Omega}{\langle u, \mathsf{cj}(b,l) \cdot v, a, s\rangle \to \mathcal{B}(b)(s)}$$

[Jsr1] $$\frac{u \cdot \mathsf{jsr}(l) \cdot v = x \cdot l \cdot y}{\langle u, \mathsf{jsr}(l) \cdot v, a, s\rangle \to \langle x, l \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(l), v\rangle, s\rangle}$$

[Jsr2] $\langle u, \mathsf{jsr}(l) \cdot v, a, s\rangle \to$ "StackOverflow"

[Ret1] $\langle u, \mathsf{ret} \cdot v, a \cdot \langle x, y\rangle, s\rangle \to \langle x, y, a, s\rangle$

[Ret2] $\langle u, \mathsf{ret} \cdot v, \varepsilon, s\rangle \to$ "EmptyStack"

[Label] $\langle u, l \cdot v, a, s\rangle \to \langle u \cdot l, v, a, s\rangle$

[Stop] $\langle u, \varepsilon, a, s\rangle \to s$

Table 7.1. Operational semantics of the assembly language.

the u component. [Asg2] applies if evaluation of e fails in the current state; in this case the failure value $\mathcal{E}(e)(s)$ is just propagated. [Cj1] describes that a conditional jump $\mathsf{cj}(b,l)$ is not taken if b evaluates to tt in the current state: In this case control is simply transferred to the subsequent instruction. If b evaluates to ff, rule [Cj2] applies and the control is transferred to label l, the position of which is determined by the premise $u \cdot \mathsf{cj}(b,l) \cdot v = x \cdot l \cdot y$, and finally [CJ3] propagates errors resulting from evaluation of b. [Jsr1] is concerned with a subroutine jump to label l. Similarly to rule [Cj2], control is transferred to label l. However, the machine also has to store the return address. This is modeled by $\langle u \cdot \mathsf{jsr}(l), v\rangle$ being pushed onto the symbolically modeled return stack a. If execution subsequently reaches a ret instruction, execution of $\langle u \cdot \mathsf{jsr}(l), v\rangle$ is resumed as specified by [Ret1]. A processor with finite memory will not always be able to stack a return address when executing a jsr instruction. We model this by rule [Jsr2] that allows the machine to spontaneously report "StackOverflow". Of course, in an actual processor the choice between regular stacking and overflow will be mutually exclusive and not just non-deterministic as in our model. This could be modeled by furnishing [Jsr2] by a premise *StackFull* and [Jsr1] by a premise $\neg$*StackFull*, where *StackFull* is a (complicated) condition depending on the current state of the machine. However, such a more concrete model serves no purpose for the present scenario which is primary intended to show the elegance of wrp-based reasoning. Finally, [Ret2] reports an error if a ret instruction is executed on an empty return stack, [Label] allows to skip labels and, for technical reasons, [Stop] propagates the resulting state if nothing is left to execute, i.e. ε plays the role of a halt-command in this case.

The evaluation of m in state s starts in the initial configuration $\langle \varepsilon, m, \varepsilon, s\rangle$, i.e. with the first instruction of m and with an empty stack. Execution terminates regularly in state s' if a configuration of the form $\langle u, \varepsilon, a, s'\rangle$ is reached such that [Stop] applies; other possible outcomes from s are reachable irregular- (error-) configurations ω and also ∞ if there is an infinite sequence of transitions starting in $\langle \varepsilon, m, \varepsilon, s\rangle$. Note that ∞ is also a reachable error configuration due to the definition of the evaluation functions $\mathcal{E}$ and $\mathcal{B}$ which model concrete evaluations that may spontaneously diverge.

7.3.3 wrp-semantics of the assembly language

The operational semantics just given should be kept for the ultimative reference: The basic description of the language which is firstly the most precise, clear and transparent definition and secondly very close to what the user has in mind and understands. For reasoning and, in particular, verification purposes, however, this description is far too granular. Each proof on this level will be clumsy and error prone just because of the step-wise definition. Thus, as motivated before, we are going to abstract from the basic operational semantics towards a more tractable but even rich predicate transformer semantics in terms of wrp. Of course, since the language considered here has no structure itself we can hardly expect structured rules concerning single commands. Nevertheless we like to elaborate sufficient rules for commands in some context that allow to reason about the behavior of programs on an abstract level.

To stress the problems one is faced with we present the general proceeding and start naively. Based on the intuition given above, for each program $m \in CMP$ a relational semantics $R(m)$ could be defined by

$$R(m) = \{(s, \sigma) \mid \langle \varepsilon, m, \varepsilon, s\rangle \rightsquigarrow \sigma\} \ ,$$

where '$\rightsquigarrow$' denotes the path-oriented relation capturing finite and infinite computations from Sect. 7.1. Relation $R(m)$ itself would give rise to a family of predicate transformers $\mathsf{wrp}_A.m$. Although totally well-defined, there is, however, a problem associated to reasoning with $\mathsf{wrp}_A.m$: It is known only with reference to the operational semantics. Thus, if we want to prove something about a program m, e.g. that it implements a source program π, we are forced to reason on base of the operational semantics. While the operational semantics provides a clear and transparent description of the semantics of assembly code (we strongly hope the reader can appreciate this), it is rather clumsy for reasoning purposes and we would prefer to reason on a more abstract level. The idea would be to derive sufficiently strong laws about $\mathsf{wrp}_A.m$ from the operational semantics first; afterwards we would use just these laws in our reasoning without a direct access to the operational semantics.

Unfortunately, this approach fails for $\mathsf{wrp}_A.m$: Only very weak laws can be established. The main problem is that the behavior of jump and jump-subroutine instructions cannot be adequately described without having context information available. We, therefore, work with a semantics of machine programs that takes the sequential context as well as the stack context into account.

We define, for $\langle u, v\rangle \in PMP$ and $a \in PMP^*$,

$$R(u,v,a) \stackrel{\text{def}}{=} \{(s,\sigma) \mid \langle u,v,a,s\rangle \rightsquigarrow \sigma\} \ ,$$

which looks quite similar to the relational semantics given above but which has the advantage that it allows to reason about isolated instructions in an arbitrary context whereas the one above only considers entire programs; the resulting wrp-rules are more generally applicable.

Again, this definition induces a family of predicate transformers, namely $\mathsf{wrp}_A.(u,v,a)$, and it is this family that we are using in our reasoning. We can, however, define resp. regain

$$R(m) \stackrel{\text{def}}{=} R(\varepsilon,m,\varepsilon) \quad \text{and} \quad \mathsf{wrp}_A.m \stackrel{\text{def}}{=} \mathsf{wrp}_A.(\varepsilon,m,\varepsilon) \ ,$$

but as just mentioned this degenerates to a more readable notation without any further use.

Now, it is time to benefit from the observations made in Sect. 7.1. The operational semantics '$\rightarrow$' of the assembly language given above is exactly in the shape of the transition relation which gave rise to the fixpoint-characterization of the wrp-transformers. Here, (u,v,a) is the control-flow component and s is the state component. Instead of deriving the aspired laws about $\mathsf{wrp}_A.(u,v,a)$ from the relational level they can be obtained from simpler fixpoint reasoning, from fixpoint unrolling to be precise. For assignments this yields

$$\begin{array}{ll}
 & s \in \mathsf{wrp}_A.(u, \mathsf{asg}(x,e)\cdot v, a).\psi \\
\Longleftrightarrow & \quad \{\mathsf{wrp} \text{ is a fixpoint of } D \text{ (Theorem 7.1.1)}\} \\
 & s \in D.\mathsf{wrp}.A.(u, \mathsf{asg}(x,e)\cdot v, a).\psi \\
\Longleftrightarrow & \quad \{\text{Definition of } D\} \\
 & \neg(\langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \rightarrow \Omega \setminus A) \ \wedge \\
 & (\forall s' : \ \langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \rightarrow s' : \ s' \in \psi) \ \wedge \\
 & (\forall \langle u', v', a', s'\rangle : \ \langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \rightarrow \langle u', v', a', s'\rangle : \\
 & \qquad\qquad s' \in \mathsf{wrp}_A.(u', v', a').\psi) \\
\Longleftrightarrow & \quad \{\text{Operational Semantics (Table 7.1) and some logic}\} \\
 & \mathcal{E}(e)(s) \in A \ \vee \\
 & (\mathcal{E}(e)(s) \in \Sigma \wedge s\{x \mapsto \mathcal{E}(e)(s)\} \in \mathsf{wrp}_A.(u \cdot \mathsf{asg}(x,e), v, a).\psi) \\
\Longleftrightarrow & \quad \{\text{Substitution}\} \\
 & \mathcal{E}(e)(s) \in A \ \vee \\
 & (\mathcal{E}(e)(s) \in \Sigma \wedge s \in (\mathsf{wrp}_A.(u \cdot \mathsf{asg}(x,e), v, a).\psi)[e/x]) \\
\Longleftrightarrow & \quad \{\text{Basic predicates} \\
 & \qquad \text{and assignment predicate transformer}\} \\
 & s \in (x :=_A e)(\mathsf{wrp}_A.(u \cdot \mathsf{asg}(x,e), v, a).\psi) \\
\Longleftrightarrow & \quad \{\text{Composition of predicate transformers}\} \\
 & s \in (\ (x :=_A e)\ ;\ \mathsf{wrp}_A.(u \cdot \mathsf{asg}(x,e), v, a)\ ).\psi \ ,
\end{array}$$

which proves

Lemma 7.3.1 (Asg-wrp).

$$\mathsf{wrp}_A.(u, \mathsf{asg}(x,e)\cdot v, a) \;=\; (x :=_A e)\ ;\ \mathsf{wrp}_A.(u\cdot \mathsf{asg}(x,e), v, a)\ )$$

□

The other instructions allow similar calculations though some of them seem to be more complicated. Let us therefore have a look at, e.g., the conditional jump. Here, the little calculation

$$\begin{array}{ll}
 & s \in \mathsf{wrp}_A.(u, \mathsf{cj}(b,l)\cdot v, a).\psi \\
\Longleftrightarrow & \quad \{\mathsf{wrp} \text{ is a fixpoint of } D \text{ (Theorem 7.1.1)}\} \\
 & s \in D.\mathsf{wrp}.A.(u, \mathsf{cj}(n,l)\cdot v, a).\psi \\
\Longleftrightarrow & \quad \{\text{Definition of } D\} \\
 & \neg(\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to \Omega \setminus A) \ \wedge \\
 & (\forall s' :\ \langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to s' :\ s' \in \psi) \ \wedge \\
 & (\forall \langle u', v', a', s'\rangle :\ \langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to \langle u', v', a', s'\rangle : \\
 & \qquad\qquad s' \in \mathsf{wrp}_A.(u', v', a').\psi) \\
\Longleftrightarrow & \quad \{\text{Operational Semantics (Table 7.1), some logic,} \\
 & \quad\ \text{assume } u\cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y\} \\
 & \mathcal{B}(b)(s) \in A \ \vee \\
 & (\mathcal{B}(b)(s) = \mathsf{tt} \wedge s \in \mathsf{wrp}_A(u\cdot \mathsf{cj}(b,l), v, a).\psi) \ \vee \\
 & (\mathcal{B}(b)(s) = \mathsf{ff} \wedge s \in \mathsf{wrp}_A(x, l\cdot y, a).\psi) \\
\Longleftrightarrow & \quad \{\text{Notational conventions from Sect. 7.2}\} \\
 & s \in (\mathsf{wrp}_A.(u\cdot \mathsf{cj}(b,l), v, a) \lhd b/A \rhd \mathsf{wrp}_A.(x, l\cdot y, a)).\psi \ ,
\end{array}$$

proves

Lemma 7.3.2 (Cj-wrp). If $u\cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y$, then

$$\mathsf{wrp}_A.(u, \mathsf{cj}(b,l)\cdot v, a) \;=\; \mathsf{wrp}_A(u\cdot \mathsf{cj}(b,l), v, a) \lhd b/A \rhd \mathsf{wrp}_A(x, l\cdot y, a)\ .$$

□

The behavior of the other instructions can be described by means of similar laws which are collected in Table 7.2. These laws allow algebraic calculations with wrp_A. The benefit over operational reasoning is that there is no longer an explicit state argument and that we can take advantage from the fact that '$\geq$' is an order. These laws still allow to perform a kind of symbolic execution of assembly programs but on a more abstract, i.e. state-free, level and it is noteworthy that each law directly corresponds to a correctness property.

All these laws can be strengthened to equalities as wrp is a fixpoint of D, see the proofs of Lemmas 7.3.1 and 7.3.2. We state them as inequalities in order to stress

[Asg-wrp]	$\mathsf{wrp}_A.(u, \mathsf{asg}(x,e)\cdot v, a) \geq$
	$\quad (x :=_A e)\ ;\ \mathsf{wrp}_A.(u\cdot \mathsf{asg}(x,e), v, a)$
[Cj-wrp]	$\mathsf{wrp}_A.(u, \mathsf{cj}(b,l)\cdot v, a) \geq$
	$\quad \mathsf{wrp}_A.(u\cdot \mathsf{cj}(b,l), v, a) \lhd b/A \rhd \mathsf{wrp}_A.(x, l\cdot y, a)$,
	$\quad\quad$ if $u\cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y$
[Goto-wrp]	$\mathsf{wrp}_A.(u, \mathsf{goto}(l)\cdot v, a) \geq \mathsf{wrp}_A.(x, l\cdot y, a)$,
	$\quad\quad$ if $u\cdot \mathsf{goto}(l)\cdot v = x\cdot l\cdot y$
[Jsr-wrp]	$\mathsf{wrp}_A.(u, \mathsf{jsr}(l)\cdot v, a) \geq \mathsf{wrp}_A.(x, l\cdot y, a\cdot\langle u\cdot \mathsf{jsr}(l), v\rangle)$,
	$\quad\quad$ if $u\cdot \mathsf{jsr}(l)\cdot v = x\cdot l\cdot y$ and "StackOverflow" $\in A$
[Ret-wrp]	$\mathsf{wrp}_A.(u, \mathsf{ret}\cdot v, a\cdot\langle x, y\rangle) \geq \mathsf{wrp}_A.(x, y, a)$
[Label-wrp]	$\mathsf{wrp}_A.(u, l\cdot v, a) \geq \mathsf{wrp}_A.(u\cdot l, v, a)$
[Term-wrp]	$\mathsf{wrp}_A.(u, \varepsilon, a) \geq Id$

Table 7.2. wrp-laws for the assembly language.

that just one direction is needed in the following.[8] If, however, [Jsr1] and [Jsr2] are furnished with a condition *StackFull* as discussed above, the refinement inequality stated in [Jsr-wrp] becomes proper because execution of jsr on the left hand side would definitely lead to the acceptable error "StackOverflow" if *StackFull* holds. Therefore, the predicate transformer on the left hand side would succeed for all states satisfying *StackFull*, irrespective of the post-condition, while the right hand side may fail.

Note that the premise "StackOverflow" $\in A$ of the law [Jsr-wrp] is essential. Let us discuss what happens if "StackOverflow" is considered unacceptable ("StackOverflow" $\notin A$). Then we have $\mathsf{wrp}_A.(u, \mathsf{jsr}(l)\cdot v, a) = \bot$ as a consequence of [Jsr2]. This means that jsr cannot be used to implement any non-trivial statement. If the more precise operational model with a *StackFull* predicate is used, $\mathsf{wrp}_A.(u, \mathsf{jsr}(l)\cdot v, a)$ is better than $\bot$ but any non-trivial approximation will involve the *StackFull* predicate. This would force us to keep track of the storage requirements when we head for a verified compilation. As the recursion depth of programs is in general not computable, we could not justify the translation of arbitrary recursive procedures.

7.4 A Simple High-Level Language

As a prototypic instance of a high-level language we consider the following WHILE-language with parameterless (i.e. without local parameters) and nested procedures. Such a language serves as an adequate means to study the very basics of the control-flow aspects of ALGOL-like programming languages.

[8] The *derived* laws presented in Fig. 7.2 can also be taken for a *definition* of the instructions' semantics. They express, in a refinement-algebraic style, what the instruction are supposed to effect *at least* and they leave effects unspecified that are not further documented. In this sense the inequalities are safe abstractions. This kind of axiomatic semantics for an assembly language is, for instance, the reference level in [61]. See also the appendix.

7.4.1 Syntax

We define the set of programs, Π, by the following grammar. In order to distinguish programs clearly from corresponding semantic predicate transformers we use an abstract kind of syntax.

$$\begin{array}{rcl} \pi & ::= & \mathsf{skip} \mid \mathsf{assign}(x,e) \mid \mathsf{seq}(\pi_1,\pi_2) \mid \mathsf{if}(b,\pi_1,\pi_2) \mid \\ & & \mathsf{while}(b,\pi) \mid \mathsf{call}(p) \mid \mathsf{blk}(p,\pi_p,\pi_b) \end{array}$$

In this grammar, x ranges over the variables in Var, b and e over $BExpr$ and $Expr$ respectively, and p over a set $ProcName$ of procedure identifiers.

The command $\mathsf{blk}(p,\pi_p,\pi_b)$ represents a block in which a (possibly recursive) local procedure p with body π_p is declared. Here, π_b is the body of the block; it might call p as well as more globally defined procedures. Note that nesting of procedure declarations and even blockwise re-declaration (introduction of procedures with names which are already in use) is allowed. The presented exposition straightforwardly generalizes to blocks in which a system of mutually recursive procedures can be declared instead of just one single procedure (this will be explained also in the semantics). We refrained from treating this more general case only as it burdens the notation a bit here and quite a lot in the remainder (when reasoning about semantics, see below, and translations, see Sect. 8.2) without bringing more insight. The intuitive semantics of the other commands should be clear from their name: skip, assignment, sequential composition, conditional, loop and procedure call.

7.4.2 Basic operational semantics

Again we start with an operational semantics – actually this is a *structural operational semantics* [71] because it is defined on the structure of programs – which is based on a transition relation on high-level language configurations, $\rightarrow \subseteq \Gamma_\Pi \times (\Gamma_\Pi \cup \Sigma \cup \Omega)$. Here, a configuration is of type $\Gamma_\Pi \stackrel{\text{def}}{=} PDict \times \Pi \times \Sigma$ where $PDict$ denotes the set of *stacks* of syntactic *procedure dictionaries*

$$\rho \in (ProcName \xrightarrow{\text{fin.}} \Pi) \ ,$$

each of which is intuitively intended to relate finitely many, i.e. one in this case, procedure names to – current – corresponding bodies.

Obviously, a dictionary is needed for describing the semantics of procedure calls. As nesting and blockwise re-declarations are allowed and on account of being a bit more flexible we are nevertheless going to use stacks of dictionaries because in so doing the recipe presented here generalizes also to scenarios in which procedures are furnished with local parameters. However, we will aim at a so-called *static scoping semantics* so let us first of all explain what we intend to model.

Intuitively, for a specific call, not always the youngest declarations in scope are to be taken but the ones which belong to the smallest block that defines the considered procedure and which surrounds the particular call in question; this should become clear in the following little example. Consider, for instance, the program

$$\mathsf{blk}(p, \pi_1, (\mathsf{blk}(q, \mathsf{call}(p), (\mathsf{blk}(p, \pi_2, \mathsf{seq}(\mathsf{call}(q), \mathsf{call}(p)))))))\ .$$

The actual body of this program is the sequential composition of two procedure calls. Executing the first, the call of q, means to execute q's body and thus the call of procedure p. Of course – at least this is what is to be modeled – here p is bound to π_1 because the smallest block introducing p which surrounds this call of p is the outermost. Furthermore the interior block, i.e. the one which defines q, shall be allowed to call *at most* more globally defined procedures; note that p was bound to π_2 when q was called. If π_1 delivers a regular result the computation should continue with a second call of procedure p. Now, p is bound to π_2 because here the innermost block is the smallest block introducing p that surrounds this call of p. The outermost declaration of p is masked behind the re-declaration. In contrast to this proceeding a so-called *dynamic scoping semantics* would always use the current bindings in scope. In the above example this results in a very different behavior: The first call of p would refer to π_2 because, as just mentioned, when the call of q is executed the youngest declaration of p is the innermost.

To model static scoping in a purely structural operational semantics, i.e. without usage of any closures in the actual definition, it seems unavoidable to make all bindings distinguished while the program is executed; actually this resembles the implementation using so-called frames or activation records in the real world (see the classic text books). Hence, whenever a blk-command, say $\mathsf{blk}(p, \pi_p, \pi)$, is to be executed all occurrences of p inside $\mathsf{blk}(p, \pi_p, \pi)$ will be replaced by a fresh identifier, the stack of dictionaries used so far will get a new entry assigning this fresh identifier to the modified procedure body, and execution continues with the modified body of the block.

For the sake of accuracy we make the following conventions. Elements of stacks of dictionaries *PDict* are ranged over by $\boldsymbol{\rho}$, i.e. a bold-face 'ρ'. The empty dictionary, i.e. the everywhere undefined mapping, is given by $\emptyset$, the empty stack by ε. Concatenation of stacks is denoted by an infix dot, '$\cdot$', and $\rho_i \in \boldsymbol{\rho}$ means that $\boldsymbol{\rho}$ contains dictionary ρ_i, i.e. $\rho_i \in \boldsymbol{\rho}$ iff $\boldsymbol{\rho} = \rho_1 \cdot \ldots \cdot \rho_i \cdot \ldots \cdot \rho_n$. The stack of dictionaries up to an index i is denoted by $\boldsymbol{\rho_i}$, i.e. $\boldsymbol{\rho_i} = \rho_1 \cdot \ldots \cdot \rho_i$. Furthermore, we also write $p \in \mathrm{dom}(\boldsymbol{\rho})$ if there exists a dictionary ρ with $\rho \in \boldsymbol{\rho}$ and $p \in \mathrm{dom}(\rho)$. A so-called fresh identifier must not be used so far, and at execution time no procedure body introduced before must refer to this fresh identifier. Note that programs may be non-closed, i.e. there may be calls to procedures that are not introduced until then, and only more globally declared procedures may be called. Furthermore, so-called free procedure identifiers must not be wrongly bound because otherwise calls of an undeclared procedure would have a sensible meaning. To model all this, we inductively define a binary relation "not used in" $nui \subseteq \mathit{ProcName} \times \Pi$ as follows:

$$p\ nui\ \mathsf{skip}\ ,$$
$$p\ nui\ \mathsf{assign}(x, e)\ ,$$
$$p\ nui\ \pi \implies p\ nui\ \mathsf{while}(b, \pi)\ ,$$
$$p\ nui\ \pi_1 \wedge p\ nui\ \pi_2 \implies p\ nui\ \mathsf{seq}(\pi_1, \pi_2)\ ,$$

$$p \neq q \wedge p \; nui \; \pi_1 \wedge p \; nui \; \pi_2 \implies p \; nui \; \mathsf{blk}(q, \pi_1, \pi_2) \;\; ,$$
$$p \neq q \implies p \; nui \; \mathsf{call}(q) \;\; .$$

Then a procedure identifier p is said to be *fresh w.r.t.* π iff $p \; nui \; \pi$ and for the sake of brevity we also say that p is fresh w.r.t. $\boldsymbol{\rho}$ iff

$$p \notin \mathrm{dom}(\boldsymbol{\rho}) \;\wedge\; \forall \rho \in \boldsymbol{\rho} :: \; \forall q \in \mathrm{dom}(\rho) :: \; p \; nui \; \rho(q) \;\; ,$$

note that the second conjunct does not imply the first as there may be procedures that are never called. In the sequel, stacks of dictionaries $\boldsymbol{\rho} = \rho_1 \cdot \ldots \cdot \rho_n$ will be *distinguished*, i.e. all procedure names in $\mathrm{dom}(\boldsymbol{\rho})$ are pairwisely different and no procedure body bound by a prefix stack calls a procedure which will be bound later. This property can be defined by[9]

$$\forall i : \; 1 \leq i \leq n : \; (\forall p \in \mathrm{dom}(\rho_i) :: \; p \text{ is fresh w.r.t } \boldsymbol{\rho_{i-1}}) \;\; ,$$

where we assume $\boldsymbol{\rho_0} = \emptyset$. Note that a distinguished stack $\boldsymbol{\rho_i}$ has the prefix-property that $\boldsymbol{\rho_j}$ is also distinguished for all $j < i$. Consequently, if $\boldsymbol{\rho}$ is distinguished and $p \in \mathrm{dom}(\boldsymbol{\rho})$ then there exists a unique dictionary ρ with $\rho \in \boldsymbol{\rho}$ and $p \in \mathrm{dom}(\rho)$; this particular dictionary is assumed to have index j_p, i.e. $p \in \mathrm{dom}(\boldsymbol{\rho}) \iff p \in \mathrm{dom}(\rho_{j_p})$. Finally, $\pi[q/p]$ denotes a *naive substitution* or *renaming* of p by q in π, i.e. $\pi[q/p]$ is the program that completely coincides with π except for all occurrences of p inside π which are replaced by q.[10]

The operational semantics is given in Table 7.3 where we assume given a distinguished stack of dictionaries $\boldsymbol{\rho}$. After the preparations above this semantic description is nearly intuitive and easy to understand, but let us make the following remarks. Executing a block $\mathsf{blk}(p, \pi_p, \pi)$, see the [Blk]-rules, means to execute the modified body of the block – each occurrence of p inside π is replaced by a fresh identifier – in a context (we avoid the word "environment" here as it will get a specific meaning soon) where a new entry is pushed onto the current stack of dictionaries; it assigns the fresh identifier to the modified procedure body in which again each occurrence of p is replaced by the fresh identifier. As all procedure names in $\mathrm{dom}(\boldsymbol{\rho})$ are pairwisely different at execution time, a call of a procedure p, see the [Call]-clauses, is replaced by the body – note that this body is a modified one as it entered the dictionary by a blk-command – to which the current dictionary assigns p; of course only if p is declared, otherwise an according error will emerge. Note that this proceeding straightforwardly generalizes to blocks in which more than just one procedure is introduced; a finite mapping like $\{p_1 \mapsto \pi_1, \ldots, p_n \mapsto \pi_n\}$ following the construction just mentioned is pushed onto the stack. The execution of all other commands is straightforward and assumed to be clear for the average programmer.

[9] There are other definitions of "distinguished stacks" in literature but the one presented here comes close to them and will suffice for our purposes.

[10] Readers familiar with these concerns might expect a so-called bound renaming. The operational semantics given below does not need this strong requirement because each block will be tackled in isolation using a fresh identifier and a distinguished stack of dictionaries. Therefore, we do not have to care for prevention from some violations of the bindings here, the sensible choice of fresh identifiers assures that free occurrences remain free and that bound identifiers refer to the corresponding block.

[Skip] $\dfrac{}{\langle \boldsymbol{\rho}, \mathsf{skip}, s\rangle \to s}$

[Asg1] $\dfrac{\mathcal{E}(e)(s) \in \mathit{Val}}{\langle \boldsymbol{\rho}, \mathsf{assign}(x,e), s\rangle \to s\{x \mapsto \mathcal{E}(e)(s)\}}$

[Asg2] $\dfrac{\mathcal{E}(e)(s) \in \Omega}{\langle \boldsymbol{\rho}, \mathsf{assign}(x,e), s\rangle \to \mathcal{E}(e)(s)}$

[Seq1] $\dfrac{\langle \boldsymbol{\rho}, \pi_1, s\rangle \to \langle \boldsymbol{\rho}, \pi_1', s'\rangle}{\langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1,\pi_2), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1',\pi_2), s'\rangle}$

[Seq2] $\dfrac{\langle \boldsymbol{\rho}, \pi_1, s\rangle \to s'}{\langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1,\pi_2), s\rangle \to \langle \boldsymbol{\rho}, \pi_2, s'\rangle}$

[Seq3] $\dfrac{\langle \boldsymbol{\rho}, \pi_1, s\rangle \to \omega}{\langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1,\pi_2), s\rangle \to \omega}$

[Cond1] $\dfrac{\mathcal{B}(b)(s) = \mathsf{tt}}{\langle \boldsymbol{\rho}, \mathsf{if}(b,\pi_1,\pi_2), s\rangle \to \langle \boldsymbol{\rho}, \pi_1, s\rangle}$

[Cond2] $\dfrac{\mathcal{B}(b)(s) = \mathsf{ff}}{\langle \boldsymbol{\rho}, \mathsf{if}(b,\pi_1,\pi_2), s\rangle \to \langle \boldsymbol{\rho}, \pi_2, s\rangle}$

[Cond3] $\dfrac{\mathcal{B}(b)(s) \in \Omega}{\langle \boldsymbol{\rho}, \mathsf{if}(b,\pi_1,\pi_2), s\rangle \to \mathcal{B}(b)(s)}$

[While] $\dfrac{}{\langle \boldsymbol{\rho}, \mathsf{while}(b,\pi), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{if}(b, \mathsf{seq}(\pi, \mathsf{while}(b,\pi)), \mathsf{skip}), s\rangle}$

[Blk1] $\dfrac{\begin{array}{c}p' \text{ is fresh w.r.t. } \boldsymbol{\rho},\ \pi \text{ and } \pi_p \\ \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi[p'/p], s\rangle \to \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi'[p'/p], s'\rangle\end{array}}{\langle \boldsymbol{\rho}, \mathsf{blk}(p,\pi_p,\pi), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{blk}(p,\pi_p,\pi'), s'\rangle}$

[Blk2] $\dfrac{\begin{array}{c}p' \text{ is fresh w.r.t. } \boldsymbol{\rho},\ \pi \text{ and } \pi_p \\ \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi[p'/p], s\rangle \to \sigma\end{array}}{\langle \boldsymbol{\rho}, \mathsf{blk}(p,\pi_p,\pi), s\rangle \to \sigma}$

[Call1] $\dfrac{p \in \mathrm{dom}(\boldsymbol{\rho})}{\langle \boldsymbol{\rho}, \mathsf{call}(p), s\rangle \to \langle \boldsymbol{\rho}, \rho_{j_p}(p), s\rangle}$

[Call2] $\dfrac{p \notin \mathrm{dom}(\boldsymbol{\rho})}{\langle \boldsymbol{\rho}, \mathsf{call}(p), s\rangle \to \text{“ProcUndecl”}}$

Table 7.3. Structural operational semantics of the high-level language.

The outcomes of interest are those which are "reachable" from a given configuration. A computation π starting in initial state s using a stack of dictionaries $\boldsymbol{\rho}$ is said to terminate in $\sigma \in \Sigma \cup \Omega$ iff there is a finite sequence

$$\langle \boldsymbol{\rho}, \pi, s \rangle \to \ldots \to \langle \boldsymbol{\rho}, \pi', s' \rangle \to \sigma$$

and this is denoted by $\langle \boldsymbol{\rho}, \pi, s \rangle \stackrel{+}{\to} \sigma$ for short.[11] If, otherwise, there is a computation starting in $\langle \boldsymbol{\rho}, \pi, s \rangle$ that may reach no regular or irregular outcome in finitely many steps then π is said to diverge, $\langle \boldsymbol{\rho}, \pi, s \rangle \to^{\infty}$ for short.

In the very end, evaluation of a program π starts with the empty dictionary, $\emptyset$, because otherwise undefined procedures would have a non-trivial meaning. Thus, initial states s and outcomes σ are of interest for which $\langle \emptyset, \pi, s \rangle \stackrel{+}{\to} \sigma$ and/or $\langle \emptyset, \pi, s \rangle \to^{\infty}$.

7.4.3 wrp-semantics of the high-level language

Following the ideas of Sect. 7.3 the next steps look auspicious. As relation '$\to$' is of appropriate type we feel inspired by (7.3) and specify

$$\langle \boldsymbol{\rho}, \pi, s \rangle \rightsquigarrow \sigma \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \langle \boldsymbol{\rho}, \pi, s \rangle \stackrel{+}{\to} \sigma \ \vee \ (\langle \boldsymbol{\rho}, \pi, s \rangle \to^{\infty} \ \wedge \ \sigma = \infty) \ .$$

This relation gives rise to the following definition where stacks of dictionaries are directly adopted from the operational semantics and taken as superscript.

Definition 7.4.1 (wrp-transformer, dictionary based). We specify the *dictionary based* wrp-*transformer* of program π w.r.t. A and $\boldsymbol{\rho}$ by

$$\mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\pi.\psi \ \stackrel{\text{def}}{=} \ \{ s \mid \forall \sigma : \ \langle \boldsymbol{\rho}, \pi, s \rangle \rightsquigarrow \sigma : \ \sigma \in \psi \cup A \} \ .$$

□

It is this semantics from which hopefully reasonable and facilitating laws are to be derived. Unluckily, the achievements would not be very satisfactory because the procedure mechanism is not adequately handled. The meaning of a procedure call is given by the meaning of the body but in general the latter is structurally not smaller than the call itself; execution of the body remains unavoidable in some sense and thus the semantics of a call is only known with regard to the entire execution of the program in question. Much more desirable is a strictly compositional, a denotational semantics that reflects the structure of the language and which allows to infer program properties from the parts of which it is built.

The usual and common means for defining a compositional denotational semantics for languages with procedures are *environments*

$$\eta \in \mathit{Env} \ \stackrel{\text{def}}{=} \ (\mathit{ProcName} \to \mathit{PTrans}) \ ,$$

intended to map procedure identifiers to (current) meanings of their bodies. What follows is yet another definition of the wrp-transformer, this time an environment based. The environment η is again taken as superscript and we decided to let

[11] Note that the operational semantics relates only configurations which use the same stack.

brackets play the role of braces in a variation in the sense of (7.6); this is done to prevent from misunderstandings due to the presence of too many braces and to distinct variations clearly from stack entries.

Definition 7.4.2 (wrp-transformer, environment based). The *environment based* wrp-*transformer* of program π w.r.t. A and η is inductively specified by the following equalities.

$$\begin{aligned}
\mathsf{wrp}^{\eta}_{A}.\mathsf{skip} &= Id \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{assign}(x,e) &= (x :=_A e) \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{if}(b,\pi_1,\pi_2) &= \mathsf{wrp}^{\eta}_{A}.\pi_1 \lhd b/A \rhd \mathsf{wrp}^{\eta}_{A}.\pi_2 \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1,\pi_2) &= \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}^{\eta}_{A}.\pi_2 \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b,\pi) &= \lambda\mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi} \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p,\pi_p,\pi_b) &= \mathsf{wrp}^{\eta[p\mapsto\lambda B_{p,\pi_p,A,\eta}]}_{A}.\pi_b \\
\mathsf{wrp}^{\eta}_{A}.\mathsf{call}(p) &= \eta(p)
\end{aligned}$$

Here, the semantics of a loop is defined with aid of function $\mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi}$, cf. (7.10) on p. 95, and for blocks one makes use of function $B_{p,\pi_p,A,\eta} \in (PTrans \to PTrans)$ which is defined by

$$B_{p,\pi_p,A,\eta}(X) \stackrel{\text{def}}{=} \mathsf{wrp}^{\eta[p\mapsto X]}_{A}.\pi_p \ .$$

Furthermore, as the rule-of-thumb suggests, $\lambda = \nu$ if $\infty \in A$, and $\lambda = \mu$ otherwise. □

Let us briefly comment on the procedure mechanism which is the most interesting and less common part; the other commands will also be discussed below. The environment in question is assumed to keep the adequate semantics of each declared procedure identifier (note that undeclared procedures get just any meaning so comparing this semantics with others only makes sense for particular environments). If so, the semantics of a call is just given by application of the current environment. However, this assumption has to be established by block-commands, e.g. $\mathsf{blk}(p,\pi_p,\pi_b)$, the semantics of which is the semantics of the body, π_b, using a modified environment where the newly introduced procedure p get a new binding. This new environment maps p to the semantics of its own body, π_p, where this new environment is to be used, too, and it leaves all other procedure identifiers untouched. It is important to note that the environment based semantics uses the considered program as it is, i.e. no fresh identifiers are introduced and programs are not renamed. Intuitively this proceeding, which is nicely modeled with the aid of fixpoints, is quite clear and promises to be correct with regard to the dictionary based definition but of course this has to be proved.

So far, there is no sensible relation between both definitions because dictionaries and environments are essentially different objects. Thus, the remainder is concerned with an equivalence proof, i.e. with a justification that both definitions, Def. 7.4.1 and Def. 7.4.2, coincide for the interesting cases. Operationally spoken, the outcomes of interest are such which are "reachable" from an initial state using the empty dictionary, $\emptyset$. If each occurring procedure call refers to a

declared procedure everything is fine but otherwise the acceptance of the emerging "ProcUndecl"-error, i.e. the question if $s \in \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{call}(p).\psi$ or not for $p \notin \mathrm{dom}(\boldsymbol{\rho})$, depends on whether "ProcUndecl" $\in A$ or not. In the predicate transformer setting this reflects in choosing an appropriate initial environment η_{initial} which is defined by[12]

$$\eta_{\text{initial}}(p) = \begin{cases} \top & : \text{ "ProcUndecl"} \in A \\ \bot & : \text{ "ProcUndecl"} \notin A \end{cases}$$

for all $p \in \mathit{ProcName}$. Having explained what the "interesting cases" are the remainder is devoted to the question if

$$\mathsf{wrp}^{\emptyset}_A.\pi \;=\; \mathsf{wrp}^{\eta_{\text{initial}}}_A.\pi$$

as intended or not, cf. Theorem 7.4.3 at the end of this chapter.

7.4.4 Equivalence of operational and denotational semantics

Before going into details of an actual equivalence proof we start to uncover and exploit some of the basic properties concerning the dictionary based and the environment based wrp-transformers.

Algebraic laws for $\mathsf{wrp}^{\boldsymbol{\rho}}_A$. The dictionary based wrp-transformers enjoy some algebraic properties which read very similar to the definition of the environment based wrp-transformer. They will play an important role in a half of the actual equivalence proof so let us collect the needed ones in this paragraph. Luckily, we can benefit from Sect. 7.1 in large amounts. Letting $(\boldsymbol{\rho}, \pi)$ play the role of the control-flow component, it follows from Theorems 7.1.3 and 7.1.2 that $\mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi = \lambda D.A.(\boldsymbol{\rho}, \pi)$ with $\lambda = \mu$ if $\infty \notin A$ and $\lambda = \nu$ if $\infty \in A$, and it turns out to be worth while to unroll the fixpoint resp. to argue pointwise a bit. Some precise looks at the operational semantics presented in Table 7.3 and some careful applications of function D, cf. (7.4) on p. 89, to λD itself bring to light the following observations where similarities to Def. 7.4.2 are expected and welcome.

The first laws are rather obvious and a direct consequence of some fixpoint unrolling and the operational semantics. A skip-command has no effect on states so in terms of predicate transformers it is represented by the identity, thus

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{skip} \;=\; \mathit{Id} \;\;.$$

Assignments evaluate an expression which either fails or succeeds; in the latter case the state is varied accordingly, i.e.

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{assign}(x.e) \;=\; (x :=_A e) \;\;.$$

Not surprisingly, the weakest relative precondition of a conditional is given by the weakest relative precondition of the branch that is chosen depending on the evaluation of the guard yielding `tt` or `ff`. As evaluation of the guard may fail itself one obtains

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{if}(b, \pi_1, \pi_2) \;=\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_1 \lhd b/A \rhd \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_2 \;\;.$$

[12] Note that η_{initial} is defined subject to A.

The fact that the used stack of dictionaries is distinguished allows a simple operational step and this transfers to the predicate transformer setting as follows. If $\boldsymbol{\rho}$ is distinguished then

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{call}(p) \;=\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\rho_{j_p}(p)$$

if $p \in \mathrm{dom}(\boldsymbol{\rho})$ and

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{call}(p) \;=\; \begin{cases} \top & : \text{ “ProcUndecl”} \in A \\ \bot & : \text{ “ProcUndecl”} \notin A \end{cases}$$

otherwise.
For the seq-command one has to argue differently. The weakest precondition of a sequential composition equals the weakest precondition of the first component establishing the weakest precondition of the second, i.e.

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{seq}(\pi_1,\pi_2) \;=\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_1 \;;\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_2 \;\;,$$

but we will only use the inequalities

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{seq}(\pi_1,\pi_2) \;\geq\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_1 \;;\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_2$$

if $\infty \notin A$ and

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{seq}(\pi_1,\pi_2) \;\leq\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_1 \;;\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_2$$

for the case $\infty \in A$. The latter formulas can be shown by some pointwise reasoning at best. It is advisable to collect some laws about steps and termination similar to the ones that will be presented in a while. Then, for the first a direct proof succeeds, and for the second we recommend to prove the contraposition because in so doing it suffices to focus on finite paths. (To prove the missing inequations a fixpoint induction on function D – similar to the proof of Lemma 8.2.3 in the next chapter – added by some auxiliary results is appropriate.) We refrain from a further discussion as the mentioned equality is rather credible and because most of the needed rules will come across in similar situations soon.
A likewise reasoning establishes an according law for blk-commands, i.e.

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p,\pi_p,\pi_b) \;=\; \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_b[p'/p] \;\;,$$

if $\boldsymbol{\rho}$ is distinguished and if p' is fresh w.r.t. $\boldsymbol{\rho}$, π_p and π_b. However, again we will only use the inequalities

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p,\pi_p,\pi_b) \;\geq\; \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_b[p'/p]$$

in the case $\infty \notin A$ resp.

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p,\pi_p,\pi_b) \;\leq\; \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_b[p'/p]$$

if $\infty \in A$.
Based on the laws concerning seq, if and skip it can be easily shown that $\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi_1)$ is a fixpoint of $\mathcal{W}_{b,\mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi_1}$ but this fact will not be used literally. It suffices to observe that

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi) \;=\; \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{if}(b,\mathsf{seq}(\pi,\mathsf{while}(b,\pi)),\mathsf{skip})$$

which is again a direct consequence of fixpoint unrolling and the operational semantics.

The benefits of fresh identifiers. The operational semantics deals with fresh procedure identifiers and renamed programs when executing blocks. The denotational semantics on the other side overwrites existing entries and uses the bodies as they are. Thus, in order to be able to compare both semantics it is advisable to reason about the effect of fresh variables. The main issues of this paragraph are two renaming theorems which are seemingly obvious and well known.[13] On account of being halfway self contained we nevertheless like to prove one of them because all similar versions we know of do not transfer to our present scenario directly or at least easily. The second has an operational flavor and fits to common renaming theorems better so we omit the clumsy proof. (The interested reader is referred to [42] which considers a most general setting.)

For the sake of comprehension and modularity we begin with the following little helpers.

Lemma 7.4.1 (Cancel nested variations). For all p, π, A, η, f:

$$\eta[p \mapsto f][p \mapsto \lambda B_{p,\pi,A,\eta[p\mapsto f]}] \;=\; \eta[p \mapsto \lambda B_{p,\pi,A,\eta}] \;\;.$$

Proof. Obviously, it suffices to consider the involved environments applied to the identifier p such that it remains to see that

$$\lambda B_{p,\pi,A,\eta[p\mapsto f]} = \lambda B_{p,\pi,A,\eta} \;\;.$$

This equality represents two equalities depending on the choice of λ, each of which degenerates to two inequalities. For, e.g., $\lambda = \nu$ and '$\geq$' we may calculate

$$\begin{array}{ll} & \nu B_{p,\pi,A,\eta[p\mapsto f]} \geq \nu B_{p,\pi,A,\eta} \\ \Longleftarrow & \quad \{\text{Induction rule}\} \\ & B_{p,\pi,A,\eta[p\mapsto f]}(\nu B_{p,\pi,A,\eta}) \geq \nu B_{p,\pi,A,\eta} \\ \Longleftrightarrow & \quad \{\text{Unroll right fixpoint and definition of } B\} \\ & \mathsf{wrp}_A^{\eta[p\mapsto f][p\mapsto \nu B_{p,\pi,A,\eta}]}.\pi \geq \mathsf{wrp}_A^{\eta[p\mapsto \nu B_{p,\pi,A,\eta}]}.\pi \;\;, \end{array}$$

and in the last line the environments are obviously equal. An analogous calculation for the three remaining cases completes the proof.
□

The observation below looks sensible but it is not quite in the shape we like to use it. However, it is the key to the actual renaming theorem.

Lemma 7.4.2 (Renaming Lemma, environment's view). For all η, A, π, p, q, f: If q is fresh w.r.t. π then

$$\mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi \;=\; \mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi[q/p] \;\;.$$

Proof. It obviously suffices to assume $q \neq p$. Then a structural induction is performed. The cases skip and $\mathsf{assign}(x, e)$ are clear by definition, and for $\mathsf{seq}(\pi_1, \pi_2)$ and $\mathsf{if}(b, \pi_1, \pi_2)$ application of the induction hypothesis succeeds. For the loop

[13] Recall that we have a slightly different understanding of a renaming; this notion refers to a naive substitution and not to a bound renaming in the classical sense.

$\mathsf{while}(b, \pi_1)$ a "quadruplicate" application of the structural induction suffices in the sense that, e.g.,

$$
\begin{aligned}
& \mathcal{W}_{b,\mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi_1}(\mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{while}(b, \pi_1[q/p])) \\
= \quad & \{\text{Defintion of } \mathcal{W}\} \\
& (\mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi_1 \ ; \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{while}(b, \pi_1[q/p])) \lhd b/A \rhd Id \\
= \quad & \{\text{Definition of the loop's semantics, assume } \infty \notin A\} \\
& (\mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi_1 \ ; \mu\mathcal{W}_{b,\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p]}) \lhd b/A \rhd Id \\
= \quad & \{\text{Structural hypothesis applied to } \pi_1\} \\
& (\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p] \ ; \mu\mathcal{W}_{b,\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p]}) \lhd b/A \rhd Id \\
= \quad & \{ \text{ Defintion of } \mathcal{W}\} \\
& \mathcal{W}_{b,\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p]}(\mu\mathcal{W}_{b,\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p]}) \\
= \quad & \{\text{Roll the fixpoint}\} \\
& \mu\mathcal{W}_{b,\mathsf{wrp}_A^{\eta[q\mapsto f]}.\pi_1[q/p]} \\
= \quad & \{\text{Definition of the loop's semantics, assume } \infty \notin A\} \\
& \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{while}(b, \pi_1[q/p]) \ ,
\end{aligned}
$$

which implies

$$\mu\mathcal{W}_{b,\mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi_1} \leq \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{while}(b, \pi_1[q/p])$$

and thus the claim for the case $\infty \notin A$ and '$\leq$'. The remaining three cases are completely analogous. Now suppose $\pi = \mathsf{call}(r)$ for an arbitrary $r \in \mathit{ProcName}$. If $r = p$ then $\pi[q/p] = \mathsf{call}(q)$ and thus

$$
\begin{aligned}
& \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{call}(p) \\
= \quad & \{\text{Definition}\} \\
& \eta[p \mapsto f](p) \\
= \quad & \{\text{Application}\} \\
& f \\
= \quad & \{\text{Reverse first two steps}\} \\
& \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{call}(q) \ .
\end{aligned}
$$

In the case $r \neq p$ we note that also $r \neq q$ as q is fresh w.r.t. π. Then

$$
\begin{aligned}
& \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{call}(r) \\
= \quad & \{\text{Definition}\} \\
& \eta[p \mapsto f](r) \\
= \quad & \{p \neq r \neq q\} \\
& \eta[q \mapsto f](r) \\
= \quad & \{\text{Definition}\} \\
& \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{call}(r) \ .
\end{aligned}
$$

The most interesting case is $\pi = \mathsf{blk}(r, \pi_r, \pi_b)$ for an arbitrary $r \in \mathit{ProcName}$ and some programs π_p, π_b. Again, note that $r \neq q$ as q is assumed to be fresh w.r.t. π. In a first case, $r \neq p$,

$$
\begin{array}{ll}
 & \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{blk}(r, \pi_r, \pi_b) \\
= & \quad \{\text{Definition}\} \\
 & \mathsf{wrp}_A^{\eta[p\mapsto f][r\mapsto \lambda B_{r,\pi_r,A,\eta[p\mapsto f]}]}.\pi_b \\
= & \quad \{\text{Rearrange variations, } r \neq p\} \\
 & \mathsf{wrp}_A^{\eta[r\mapsto \lambda B_{r,\pi_r,A,\eta[p\mapsto f]}][p\mapsto f]}.\pi_b \\
= & \quad \{\text{Structural hypothesis applied to } \pi_b \\
 & \qquad \text{and } \eta[r \mapsto \lambda B_{r,\pi_r,A,\eta[p\mapsto f]}]\} \\
 & \mathsf{wrp}_A^{\eta[r\mapsto \lambda B_{r,\pi_r,A,\eta[p\mapsto f]}][q\mapsto f]}.\pi_b[q/p] \\
= & \quad \{\text{See below}\} \\
 & \mathsf{wrp}_A^{\eta[r\mapsto \lambda B_{r,\pi_r[q/p],A,\eta[q\mapsto f]}][q\mapsto f]}.\pi_b[q/p] \\
= & \quad \{\text{Rearrange variations, } q \neq r\} \\
 & \mathsf{wrp}_A^{\eta[q\mapsto f][r\mapsto \lambda B_{r,\pi_r[q/p],A,\eta[q\mapsto f]}]}.\pi_b[q/p] \\
= & \quad \{\text{Definition}\} \\
 & \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{blk}(r, \pi_r[q/p], \pi_b[q/p]) \ ,
\end{array}
$$

where it remains to justify the last but two step, i.e. actually

$$\lambda B_{r,\pi_r,A,\eta[p\mapsto f]} \ = \ \lambda B_{r,\pi_r[q/p],A,\eta[q\mapsto f]} \ .$$

Again, four cases have to be considered, e.g.,

$$
\begin{array}{ll}
 & \mu B_{r,\pi_r,A,\eta[p\mapsto f]} \geq \mu B_{r,\pi_r[q/p],A,\eta[q\mapsto f]} \\
\Longleftarrow & \quad \{\text{Induction rule}\} \\
 & \mu B_{r,\pi_r,A,\eta[p\mapsto f]} \geq B_{r,\pi_r[q/p],A,\eta[q\mapsto f]}(\mu B_{r,\pi_r,A,\eta[p\mapsto f]}) \\
\Longleftrightarrow & \quad \{\text{Unroll left fixpoint and definition of } B\} \\
 & \mathsf{wrp}_A^{\eta[p\mapsto f][r\mapsto \mu B_{r,\pi_r,A,\eta[p\mapsto f]}]}.\pi_r \geq \mathsf{wrp}_A^{\eta[q\mapsto f][r\mapsto \mu B_{r,\pi_r,A,\eta[p\mapsto f]}]}.\pi_r[q/p] \\
\Longleftrightarrow & \quad \{\text{Rearrange variations, } q \neq r \neq p\} \\
 & \mathsf{wrp}_A^{\eta[r\mapsto \mu B_{r,\pi_r,A,\eta[p\mapsto f]}][p\mapsto f]}.\pi_r \geq \mathsf{wrp}_A^{\eta[r\mapsto \mu B_{r,\pi_r,A,\eta[p\mapsto f]}][q\mapsto f]}.\pi_r[q/p] \ ,
\end{array}
$$

where the last line holds by the structural hypothesis applied to π_r and $\eta[r \mapsto \mu B_{r,\pi_r,A,\eta[p\mapsto f]}]$. In a second case, namely $r = p$, we have

$$
\begin{array}{ll}
 & \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{blk}(p, \pi_r, \pi_b) \\
= & \quad \{\text{Definition}\} \\
 & \mathsf{wrp}_A^{\eta[p\mapsto f][p\mapsto \lambda B_{p,\pi_r,A,\eta[p\mapsto f]}]}.\pi_b \\
= & \quad \{\text{Cancel nested variation (Lemma 7.4.1)}\}
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{wrp}_A^{\eta[p\mapsto\lambda B_{p,\pi_r,A,\eta}]}.\pi_b \\
= & \quad \{\text{Structural hypothesis applied to } \pi_b, \text{ see below}\} \\
& \mathsf{wrp}_A^{\eta[q\mapsto\lambda B_{q,\pi_r[q/p],A,\eta}]}.\pi_b[q/p] \\
= & \quad \{\text{Reverse first two steps}\} \\
& \mathsf{wrp}_A^{\eta[q\mapsto f]}.\mathsf{blk}(q,\pi_r[q/p],\pi_b[q/p]) \ ,
\end{array}
$$

where application of the structural hypothesis in the last but one step is admissible because

$$\lambda B_{p,\pi_r,A,\eta} = \lambda B_{q,\pi_r[q/p],A,\eta} \ ,$$

which can be shown following the approved recipe, such that each of those predicate transformers can play the role of f in the structural hypothesis.
□

This renaming lemma suggests that the current environment has to be reasonably varied if programs are renamed. In the actual semantics, however, an environment is varied by block-commands in a specific way and not arbitrarily. Therefore, the actual renaming theorem below presents a practically more relevant criterion.

Theorem 7.4.1 (Renaming Theorem, environment's view). For all η, A, π_1, π_2, p, q: If q is fresh w.r.t. π_1 and π_2 then

$$\mathsf{wrp}_A^{\eta}.\mathsf{blk}(p,\pi_1,\pi_2) \ = \ \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p,\pi_1,\pi_2)[q/p] \ .$$

Proof. Suppose given η, A, π_1, π_2, p and q such that q is fresh w.r.t. π_1 and π_2. The case $p = q$ is obvious so let us assume the opposite. The claim is equivalent to

$$\mathsf{wrp}_A^{\eta[p\mapsto\lambda B_{p,\pi_1,A,\eta}]}.\pi_2 \ = \ \mathsf{wrp}_A^{\eta[q\mapsto\lambda B_{q,\pi_1[q/p],A,\eta}]}.\pi_2[q/p] \ ,$$

and by the Renaming Lemma 7.4.2 this equality holds if

$$\lambda B_{p,\pi_1,A,\eta} = \lambda B_{q,\pi_1[q/p],A,\eta} \ .$$

Again, this equality represents four inequalities depending on the choice of λ and '$\leq$' resp. '$\geq$'. Each of these combinations follows the known proceeding, e.g.,

$$
\begin{array}{ll}
& \nu B_{p,\pi_1,A,\eta} \leq \nu B_{q,\pi_1[q/p],A,\eta} \\
\Longleftarrow & \quad \{\text{Induction rule}\} \\
& \nu B_{p,\pi_1,A,\eta} \leq B_{q,\pi_1[q/p],A,\eta}(\nu B_{p,\pi_1,A,\eta}) \\
\Longleftrightarrow & \quad \{\text{Unroll left fixpoint and definition of } B\} \\
& \mathsf{wrp}_A^{\eta[p\mapsto\nu B_{p,\pi_1,A,\eta}]}.\pi_1 \leq \mathsf{wrp}_A^{\eta[q\mapsto\nu B_{p,\pi_1,A,\eta}]}.\pi_1[q/p] \ ,
\end{array}
$$

and the last line follows from the Renaming Lemma 7.4.2 again where $\nu B_{p,\pi_1,A,\eta}$ plays f's role.
□

The operational counterpart of Theorem 7.4.1 reads as shown below. The proof is omitted because it is barely illuminating in the present scenario. Let us just

hint that a careful analysis of possible operational steps respecting consistently renamed stacks of dictionaries will guide to the goal. If one believes in the mostly unproved structural laws from the previous paragraph one may also take those into account.

Theorem 7.4.2 (Renaming Theorem, dictionary's view). For all $\boldsymbol{\rho}$, A, π_1, π_2, p, q: If $\boldsymbol{\rho}$ is distinguished and if q is fresh w.r.t. π_1, π_2 and $\boldsymbol{\rho}$ then

$$\mathsf{wrp}_A^{\boldsymbol{\rho}}.\mathsf{blk}(p,\pi_1,\pi_2) \;=\; \mathsf{wrp}_A^{\boldsymbol{\rho}}.\mathsf{blk}(p,\pi_1,\pi_2)[q/p] \;\;.$$

□

Yet another effect of fresh identifiers is the following. If a program π does not mention a specific procedure identifier, say p, i.e. if p is fresh w.r.t. π, then it does not matter if the environment in question gets a new entry for p or not because p will not be called by π at all. In fact, this is a consequence of the first Renaming Theorem 7.4.1.

Lemma 7.4.3 (Upgrading η with fresh identifiers). For all π, η, A, p, f: If p is fresh w.r.t. π then

$$\mathsf{wrp}_A^{\eta}.\pi \;=\; \mathsf{wrp}_A^{\eta[p\mapsto f]}.\pi \;\;.$$

Proof by structural induction. We restrict to commands which contain procedure identifiers; for all other commands the definitions resp. the hypotheses apply. To prove the claim for $\pi = \mathsf{blk}(q,\pi_1,\pi_2)$ with $p \neq q$ and p fresh w.r.t. π_1 and π_2 we calculate as follows:

$$\begin{array}{ll}
 & \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{blk}(q,\pi_1,\pi_2) \\
= & \quad \{\text{Renaming Theorem 7.4.1 above}\} \\
 & \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{blk}(p,\pi_1[p/q],\pi_2[p/q]) \\
= & \quad \{\text{Definition}\} \\
 & \mathsf{wrp}_A^{\eta[p\mapsto f][p\mapsto \lambda B_{p,\pi_1[p/q],A,\eta[p\mapsto f]}]}.\pi_2[p/q] \\
= & \quad \{\text{Cancel nested variations}\} \\
 & \mathsf{wrp}_A^{[p\mapsto \lambda B_{p,\pi_1[p/q],A,\eta}]}.\pi_2[p/q] \\
= & \quad \{\text{Definition}\} \\
 & \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p,\pi_1[p/q]\pi_2[p/q]) \\
= & \quad \{\text{Rename again}\} \\
 & \mathsf{wrp}_A^{\eta}.\mathsf{blk}(q,\pi_1,\pi_2) \;\;.
\end{array}$$

Finally, it is obvious that

$$\mathsf{wrp}_A^{\eta}.\mathsf{call}(q) = \eta(q) = \eta[p\mapsto f](q) = \mathsf{wrp}_A^{\eta[p\mapsto f]}.\mathsf{call}(q)$$

for all q different from p; the case $q = p$ cannot occur because p is assumed to be fresh w.r.t. π.

□

A respective law holds for dictionary based wrp-transformers. Again, due to the fact that dictionaries map identifiers to programs instead of predicate transformers

the chosen fresh identifier must not be used in the procedure bodies introduced before either.

Lemma 7.4.4 (Upgrading $\boldsymbol{\rho}$ with fresh identifiers). For all π, $\boldsymbol{\rho}$, A, p and X: If $\boldsymbol{\rho}$ is distinguished and if p is fresh w.r.t. π and $\boldsymbol{\rho}$ then

$$\mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi \;=\; \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p\mapsto X\}}_A.\pi \;\;.$$

The *proof* is omitted because it would fill pages without bringing essentially more insight.
□

Sensible relationships between dictionaries and environments. Seen from the operational perspective, a call of a procedure named p, if declared, is replaced by the program that is assigned to p by $\boldsymbol{\rho}$. Thus, the semantics of a call is somehow the semantics of its body. Intuitively, the same happens seen from the view of the predicate transformers: A call of p means to apply the current environment η to p. This binding entered η when the latest block introducing p was executed and η received a new binding which assigned p to the wrp-transformer of p's body.

But as mentioned before the operational semantics deals with distinguished stacks of dictionaries, fresh identifiers and renamed bodies whereas the denotational semantics overwrites old entries and uses unmodified bodies instead. However, one of the many keys to an equivalence is the fact that old bindings do not change if the environment gets a new entry for a fresh identifier, i.e. Lemma 7.4.3, and that it does not matter if the bodies are renamed with fresh identifiers or not, see Theorems 7.4.1 and 7.4.2. Thus, as only fresh identifiers are pushed onto a distinguished stack $\boldsymbol{\rho}$ we may indeed assume – and Lemmas 7.4.5 and 7.4.6 below ensure this – that an environment η applied to an identifier p which is bound to an introducing block, i.e. for which $p \in \mathrm{dom}(\boldsymbol{\rho})$, yields the wrp-transformer of p's body, i.e. $\mathsf{wrp}^{\eta}_A.\rho_{j_p}(p)$. If otherwise p is not introduced by a preceding blk-command as yet, a "ProcUndecl"-error will emerge. From the predicate transformer perspective the acceptance of this finite error depends on whether "ProcUndecl" $\in A$ or not. We collect our expectations in the following predicate.[14]

$$\begin{array}{l}
\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \overset{\text{def}}{\Longleftrightarrow} \\
\quad \boldsymbol{\rho} \text{ is distinguished } \wedge \\
\quad \forall p \in \mathit{ProcName} :: \\
\qquad p \in \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta(p) = \mathsf{wrp}^{\eta}_A.\rho_{j_p}(p) \;\wedge \\
\qquad p \notin \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta(p) = \left\{ \begin{array}{lcl} \top & : & \text{"ProcUndecl"} \in A \\ \bot & : & \text{"ProcUndecl"} \notin A \end{array} \right.
\end{array}$$

This predicate enjoys the particular property that it can easily be instantiated by choosing $\eta = \eta_{\text{initial}}$ and $\boldsymbol{\rho} = \emptyset$, the mappings to which one refers if properties of initially given programs are to be compared (the interesting cases). The reader will not have any problems to prove

[14] The index e stands for "environment"; the cause for this notational emphasis becomes clear in a moment.

Lemma 7.4.5 (Initialize $\mathsf{bindings}_e$). For all A:

$$\mathsf{bindings}_e(\eta_{\text{initial}}, \emptyset, A) \ .$$

□

Blocks vary the stack of dictionaries resp. the environment and the result below assures that the bindings remain sensibly related under certain conditions.

Lemma 7.4.6 (Establish $\mathsf{bindings}_e$). For all $\boldsymbol{\rho}$, η, A, p, π:

$$\begin{aligned}&\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \ \wedge \ p \text{ is fresh w.r.t. } \boldsymbol{\rho} \\ &\quad \Longrightarrow \mathsf{bindings}_e(\eta[p \mapsto \lambda B_{p,\pi,A,\eta}], \boldsymbol{\rho} \cdot \{p \mapsto \pi\}, A)\end{aligned}$$

Proof. As $\boldsymbol{\rho}$ is distinguished and p is fresh w.r.t. $\boldsymbol{\rho}$, i.e. $p \notin \mathrm{dom}(\boldsymbol{\rho})$, it is obvious that $\boldsymbol{\rho} \cdot \{p \mapsto \pi\}$ is distinguished, too. Then take an arbitrary $q \in \mathit{ProcName}$. If $q \notin \mathrm{dom}(\boldsymbol{\rho} \cdot \{p \mapsto \pi\})$ then $q \notin \mathrm{dom}(\boldsymbol{\rho})$ and in particular $q \neq p$. Thus,

$$\eta[p \mapsto \lambda B_{p,\pi,A,\eta}](q) = \eta(q) = \begin{cases} \top & : \text{ “ProcUndecl”} \in A \\ \bot & : \text{ “ProcUndecl”} \notin A \end{cases}$$

follows from the assumption. If, otherwise, $q \in \mathrm{dom}(\boldsymbol{\rho} \cdot \{p \mapsto \pi\})$ then either $q \in \mathrm{dom}(\boldsymbol{\rho})$ and $q \neq p$ or $q \notin \mathrm{dom}(\boldsymbol{\rho})$ and $q = p$. In the first case it follows from the assumption that

$$\eta[p \mapsto \lambda B_{p,\pi,A,\eta}](q) = \eta(q) = \mathsf{wrp}^{\eta}_{A}.\rho_{j_q}(q) \ ,$$

and because $\rho_{j_q}(q)$ is unique we are left with showing

$$\mathsf{wrp}^{\eta}_{A}.\rho_{j_q}(q) = \mathsf{wrp}^{\eta[p \mapsto \lambda B_{p,\pi,A,\eta}]}_{A}.\rho_{j_q}(q) \ .$$

Now, as p is fresh w.r.t. $\boldsymbol{\rho}$ we particularly know that p is fresh w.r.t. $\rho_{j_q}(q)$ such that Lemma 7.4.3 applies. The second case, $p = q$, follows from

$$\begin{aligned}&\eta[p \mapsto \lambda B_{p,\pi,A,\eta}](p) \\ =& \quad \{\text{Application}\} \\ &\lambda B_{p,\pi,A,\eta} \\ =& \quad \{\text{Unroll the fixpoint and definition of } B\} \\ &\mathsf{wrp}^{\eta[p \mapsto \lambda B_{p,\pi,A,\eta}]}_{A}.\pi \\ =& \quad \{\rho_{j_p} = \{p \mapsto \pi\} \text{ as } \boldsymbol{\rho} \cdot \{p \mapsto \pi\} \text{ is distinguished}\} \\ &\mathsf{wrp}^{\eta[p \mapsto \lambda B_{p,\pi,A,\eta}]}_{A}.\rho_{j_p}(p) \ ,\end{aligned}$$

and this completes the proof.

□

However, the watchful reader might have observed that the $\mathsf{bindings}_e$ predicate relates environments to environment based wrp-transformers (this is where the index ‘e’ stems from) and in fact $\mathsf{bindings}_e$ has a counterpart which, as expected, relates environments to dictionary based wrp-transformers. To be precise, there are two counterparts because it turns out that a nice and helpful result similar to Lemma 7.4.6 above – expressing an equality which is independent of $\infty \in A$ or not – cannot be easily achieved. Fortunately, it suffices to establish two inequalities

each of which depends on the choice of fixpoints in the semantics. Therefore, we define the following two predicates:

$$
\begin{array}{l}
\mathsf{bindings}_{\mu}(\eta, \boldsymbol{\rho}, A) \stackrel{\text{def}}{\Longleftrightarrow} \\
\quad \boldsymbol{\rho} \text{ is distinguished } \wedge \\
\quad \forall p \in \mathit{ProcName} :: \\
\qquad p \in \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta(p) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\rho_{j_p}(p) \ \wedge \\
\qquad p \notin \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta(p) = \left\{ \begin{array}{lcl} \top & : & \text{“ProcUndecl”} \in A \\ \bot & : & \text{“ProcUndecl”} \notin A \end{array} \right.
\end{array}
$$

where the index μ suggests applicability in the case that divergence is kept for intolerable, i.e. where $\infty \notin A$ and consequently $\lambda = \mu$. Note that the inequality sign promises to let simpler induction rules for least fixpoints apply. Hence, the second predicate is specified as below.

$$
\begin{array}{l}
\mathsf{bindings}_{\nu}(\eta, \boldsymbol{\rho}, A) \stackrel{\text{def}}{\Longleftrightarrow} \\
\quad \boldsymbol{\rho} \text{ is distinguished } \wedge \\
\quad \forall p \in \mathit{ProcName} :: \\
\qquad p \in \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta(p) \geq \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\rho_{j_p}(p) \ \wedge \\
\qquad p \notin \mathrm{dom}(\boldsymbol{\rho}) \longrightarrow \eta_i(p) = \left\{ \begin{array}{lcl} \top & : & \text{“ProcUndecl”} \in A \\ \bot & : & \text{“ProcUndecl”} \notin A \end{array} \right.
\end{array}
$$

Again, these two predicates can easily be instantiated with an appropriate choice of an initial dictionary resp. environment.

Lemma 7.4.7 (Initialize $\mathsf{bindings}_{\lambda}$). For all A and $\lambda \in \{\mu, \nu\}$:

$$\mathsf{bindings}_{\lambda}(\eta_{\text{initial}}, \emptyset, A) \ .$$

□

Having two predicates at hand we would consequently have two versions acting as a counterpart to Lemma 7.4.6. But to prove them one is in need of further premises so these results are given at the appropriate place, actually this will be inside the induction step of Lemmas 7.4.16 and 7.4.17.

Laws about steps and termination. For the remainder it is important to notice that the blocks $\mathsf{blk}(p, \pi_p, \pi_b)$ and $\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p])$ can be identified if p' is fresh w.r.t. π_p, π_b and a stack of dictionaries in question. From the view of the predicate transformers this observation is rather clear, the Renaming Theorems 7.4.1 and 7.4.2 express this more formally. To see this for the operational semantics we like to hint the following. Suppose given two distinct procedure identifiers p', p'', both fresh w.r.t. π_p, π_b and $\boldsymbol{\rho}$. Then p'' is also fresh w.r.t. $\pi_p[p'/p]$ and $\pi_b[p'/p]$, and, e.g., the [Blk2]-rule says

$$
\begin{array}{ll}
 & \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s \rangle \to s' \\
\text{iff} & \langle \boldsymbol{\rho} \cdot \{p'' \mapsto \pi_p[p''/p]\}, \pi_b[p''/p], s \rangle \to s' \\
\text{iff} & \langle \boldsymbol{\rho} \cdot \{p'' \mapsto \pi_p[p'/p][p''/p']\}, \pi_b[p'/p][p''/p'], s \rangle \to s' \\
\text{iff} & \langle \boldsymbol{\rho}, \mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]), s \rangle \to s' \ .
\end{array}
$$

Though this little observation does not prove anything here, it nevertheless shows that both blocks may indeed be identified (and it is also part of the recipe to prove Renaming Theorem 7.4.2). We mention this particular property because we will be in need to apply structural hypotheses to consistently renamed programs and this hint justifies our understanding of $\pi_p[p'/p]$ and $\pi_b[p'/p]$ being components of $\mathsf{blk}(p, \pi_p, \pi_b)$, of course only for a p' satisfying the requirements and if properly used. For the sake of brevity this insight will be used every now and then without explicitly mentioning it.

However, in some sense, the environment based wrp-transformers are able to mimic each step that is performed on the operational level. The results presented here look similar to Lemma 7.1.2 where the wrp-transformer was rolled and unrolled in the more general scenario but one cannot profit from Sect. 7.1 directly as a part of the control-flow component changes, too.

If there exists a step from a configuration to a final regular result then this regular result is unavoidable in some sense. The next four lemmas have a somewhat "forward" flavor and so their names are furnished with this phrase.

Lemma 7.4.8 (Forward regular-termination-lemma). For all η, $\boldsymbol{\rho}$, π, A, s and s':

$$\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \ \wedge \ \langle \boldsymbol{\rho}, \pi, s\rangle \to s' \implies s \notin \mathsf{wrp}^{\eta}_{A}.\pi.\neg s'$$

Proof. One considers each rule that is applicable and performs a structural induction. For skip-commands one obviously obtains

$$s \notin \Sigma \setminus s \ = \ Id.(\Sigma \setminus s) \ = \ \mathsf{wrp}^{\eta}_{A}.\mathsf{skip}.(\Sigma \setminus s) \ .$$

For the [Asg1]-rule one has

$$\begin{array}{ll} & s \notin \mathsf{wrp}^{\eta}_{A}.\mathsf{assign}(x, e).(\Sigma \setminus s\{x \mapsto \mathcal{E}(e)(s)\}) \\ \iff & \quad \{\text{Definition}\} \\ & s \notin \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap (\Sigma \setminus s\{x \mapsto \mathcal{E}(e)(s)\})[e/x]) \\ \impliedby & \quad \{\mathcal{E}(e)(s) \in \mathit{Val} \text{ in [Asg1]}\} \\ & s \notin (\Sigma \setminus s\{x \mapsto \mathcal{E}(e)(s)\})[e/x] \\ \iff & \quad \{\text{Definition of substitution}\} \\ & s\{x \mapsto \mathcal{E}(e)(s)\} \notin (\Sigma \setminus s\{x \mapsto \mathcal{E}(e)(s)\}) \end{array}$$

which obviously holds. If the [Blk2]-rule applies, then there is a transition $\langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s\rangle \to s'$ whenever $\langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi_b[p'/p], s\rangle \to s'$, where p' is fresh w.r.t. $\boldsymbol{\rho}$, π_p and π_b. We apply the structural hypothesis to the body of the block, $\pi_b[p'/p]$, which is admissible on the accounts made before. Thus,

$$\begin{array}{ll} & s \notin \mathsf{wrp}^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}_{A}.\pi_b[p'/p].(\Sigma \setminus s') \\ \iff & \quad \{\text{Definition}\} \\ & s \notin \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p', p]).(\Sigma \setminus s') \\ \iff & \quad \{p' \text{ is fresh w.r.t. } \pi_p \text{ and } \pi_b, \text{ hence} \\ & \qquad \text{the Renaming Theorem 7.4.1 applies}\} \\ & s \notin \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p, \pi_p, \pi_b).(\Sigma \setminus s') \ , \end{array}$$

and this completes the proof.
□

Final irregular results are unavoidable in the same sense, unless they are tolerated. The very similar lemma reads like this.

Lemma 7.4.9 (Forward irregular-termination-lemma). For all η, $\boldsymbol{\rho}$, π, A, s, ω:

$$\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \ \wedge \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \omega \ \wedge \ \omega \notin A \implies s \notin \mathsf{wrp}^{\eta}_{A}.\pi.\mathtt{true}$$

Proof. We skip the [Asg2]-rule because this case is obvious and have a look at the [Seq3]-rule. Here, $\langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1, \pi_2), s\rangle \to \omega$ if $\langle \boldsymbol{\rho}, \pi_1, s\rangle \to \omega$. Again, we apply the structural hypothesis to π_1 such that

$$\begin{array}{ll} & s \notin \mathsf{wrp}^{\eta}_{A}.\pi_1.\mathtt{true} \\ \implies & \quad \{\text{Monotonicity}\} \\ & s \notin \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\mathtt{true}) \\ \iff & \quad \{\text{Definition}\} \\ & s \notin \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\mathtt{true} \ . \end{array}$$

For the [BLK2]-rule one calculates as done in the previous lemma and the [Cond3]-rule is obvious. So consider the [Call2]-rule where $\langle \boldsymbol{\rho}, \mathsf{call}(p), s\rangle \to$ "ProcUndecl" and $p \notin \mathrm{dom}(\boldsymbol{\rho})$. Then

$$s \notin \mathtt{false} \ = \ \bot.\mathtt{true} \ = \ \eta(p).\mathtt{true} \ = \ \mathsf{wrp}^{\eta}_{A}.\mathsf{call}(p).\mathtt{true}$$

where $\eta(p) = \bot$ follows from $\mathsf{bindings}_e$ because "ProcUndecl" $\notin A$ by the latter premise.
□

Not only final transitions can be reproduced but also intermediate steps. If a configuration "gets along" with its corresponding predicate then so does each successive configuration. Here the "forward" flavor comes in sight.

Lemma 7.4.10 (Forward step-lemma). For all η, $\boldsymbol{\rho}$, π, π', A, ψ, s, s':

$$\begin{array}{l} \mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \ \wedge \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \langle \boldsymbol{\rho}, \pi', s'\rangle \ \wedge \ s \in \mathsf{wrp}^{\eta}_{A}.\pi.\psi \\ \quad \implies s' \in \mathsf{wrp}^{\eta}_{A}.\pi', \psi \end{array}$$

Proof. Again, we proceed by a structural induction. The [Seq1]-rule applies if $\langle \boldsymbol{\rho}, \pi_1, s\rangle \to \langle \boldsymbol{\rho}, \pi_1', s'\rangle$. This allows to apply the structural hypothesis to π_1, i.e., in particular,

$$\forall \phi : \ s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.\phi : \ s' \in \mathsf{wrp}^{\eta}_{A}.\pi_1'.\phi \ .$$

Then it follows that

$$\begin{array}{ll} & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\psi \\ \iff & \quad \{\text{Definition}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \implies & \quad \{\text{See above, choose } \phi = \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi\} \end{array}$$

$$
\begin{array}{ll}
 & s' \in \mathsf{wrp}^{\eta}_{A}.\pi'_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\
\iff & \quad \{\text{Definition}\} \\
 & s' \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi'_1, \pi_2).\psi \quad .
\end{array}
$$

Rule [Seq2] applies whenever $\langle \boldsymbol{\rho}, \pi_1, s\rangle \to s'$ and Lemma 7.4.8 implies $s \notin \mathsf{wrp}^{\eta}_{A}.\pi_1.\neg s'$, i.e. $s \in \neg\mathsf{wrp}^{\eta}_{A}.\pi_1.\neg s'$. Then,

$$
\begin{array}{ll}
 & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\psi \\
\iff & \quad \{\text{Definition}\} \\
 & s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\
\implies & \quad \{\text{Observation above and} \\
 & \qquad \text{“Best of both worlds”, i.e. Lemma 6.1.3}\} \\
 & s \in \neg\mathsf{wrp}_{A}.\pi_1.(\neg s' \cap \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \quad .
\end{array}
$$

Let us briefly look at the postcondition in more detail. If we assume that $s' \notin \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi$, i.e.

$$
\begin{array}{ll}
 & \{s'\} \subseteq \neg\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \\
\iff & \{s'\} \cup \neg\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi = \neg\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \\
\iff & \neg(\{s'\} \cup \neg\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) = \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \\
\iff & \neg\{s'\} \cap \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi = \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \quad ,
\end{array}
$$

then the last line above implies $s \in \neg\mathsf{wrp}_{A}.^{\eta}.\mathsf{seq}(\pi_1, \pi_2).\psi$ in contradiction to the choice of s. Thus, $s' \in \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi$ as required. The [Blk2]-rule follows the known recipe; it applies if

$$\langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi_b[p'/p], s\rangle \to \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi'[p'/p], s'\rangle$$

with an identifier p' that is fresh w.r.t. $\boldsymbol{\rho}$, π_p and π_b. Lemma 7.4.6 allows to apply the structural hypothesis to $\pi_b[p'/p]$ in

$$
\begin{array}{ll}
 & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p, \pi_p, \pi_b).\psi \\
\iff & \quad \{\text{Renaming Theorem 7.4.1}\} \\
 & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]).\psi \\
\iff & \quad \{\text{Definition}\} \\
 & s \in \mathsf{wrp}^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}_{A}.\pi_b[p'/p].\psi \\
\implies & \quad \{\text{Here!}\} \\
 & s' \in \mathsf{wrp}^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}_{A}.\pi'[p'/p].\psi \\
\iff & \quad \{\text{Reverse first two steps}\} \\
 & s' \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p, \pi_p, \pi').\psi \quad ,
\end{array}
$$

note that we again identified $\mathsf{blk}(p, \pi_p, \pi_b)$ with $\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p])$ which allows to apply the structural hypothesis to renamed programs. The two rules for conditionals, [Cond1] and [Cond2], are omitted as they are not illuminating, so have a look at the loop-rule [While]. Here one calculates

$$
\begin{array}{ll}
& \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b,\pi) \\
= & \quad \{\text{Definition}\} \\
& \lambda\mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi} \\
= & \quad \{\text{Unroll Fixpoint}\} \\
& \mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi}(\lambda\mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi}) \\
= & \quad \{\text{Definitions of } \mathcal{W} \text{ and the loop's semantics}\} \\
& (\mathsf{wrp}^{\eta}_{A}.\pi \ ; \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b,\pi)) \lhd b/A \rhd Id \\
= & \quad \{\text{Definitions}\} \\
& \mathsf{wrp}^{\eta}_{A}.\mathsf{if}(b,\mathsf{seq}(\pi,\mathsf{while}(b,\pi)),\mathsf{skip}) \ ,
\end{array}
$$

so this case is obvious, too. Finally, the [Call2]-rule is nicely captured by our assumptions about $\mathsf{bindings}_e$. Since $p \in \mathrm{dom}(\boldsymbol{\rho})$ we obtain

$$
\begin{array}{ll}
& \mathsf{wrp}^{\eta}_{A}.\mathsf{call}(p) \\
= & \quad \{\text{Definition}\} \\
& \eta(p) \\
= & \quad \{\text{See } \mathsf{bindings}_e(\eta,\boldsymbol{\rho},A)\} \\
& \mathsf{wrp}^{\eta}_{A}.\rho_{j_p}(p) \ ,
\end{array}
$$

which completes the proof.
□

Altogether, the latter three Lemmas, 7.4.8, 7.4.9 and 7.4.10, allow to mimic particular computations, namely those which do not yield a desired finite result (this interpretation stems from its application, Lemma 7.4.14).

Lemma 7.4.11 (Forward sequence-lemma). For all η, $\boldsymbol{\rho}$, π, A, s, s', ω:

$$\mathsf{bindings}_e(\eta,\boldsymbol{\rho},A) \ \wedge \ \langle\boldsymbol{\rho},\pi,s\rangle \xrightarrow{+} s' \implies s \notin \mathsf{wrp}^{\eta}_{A}.\pi.\neg s'$$

and

$$\mathsf{bindings}_e(\eta,\boldsymbol{\rho},A) \ \wedge \ \langle\boldsymbol{\rho},\pi,s\rangle \xrightarrow{+} \omega \ \wedge \omega \notin A \implies s \notin \mathsf{wrp}^{\eta}_{A}.\pi.\mathtt{true}$$

Proof. Due to symmetry we discuss the first claim only. Consider a finite path, say

$$\langle\boldsymbol{\rho},\pi_1,s_1\rangle \to \ldots \to \langle\boldsymbol{\rho},\pi_n,s_n\rangle \to s' \ ,$$

where $\pi_1 = \pi$, $s_1 = s$ and $n \geq 1$. We note that $\mathsf{bindings}_e(\eta,\boldsymbol{\rho},A)$ holds for all intermediate steps because it is assumed to hold at the beginning and the transition relation '$\to$' only deals with configurations where $\boldsymbol{\rho}$ is used on both sides. Thus, we will freely assume $\mathsf{bindings}_e$ to holds whenever this is needed. By the forward regular-termination-lemma (Lemma 7.4.8) the final step is unavoidable, i.e. $s_n \notin \mathsf{wrp}^{\eta}_{A}.\pi_n.(\Sigma \setminus s')$. Shunting both wrp-terms in the forward step-lemma 7.4.10 and inductive, i.e. $n$fold, application of the same lets us agree that the predicate transformers cannot be prevented from reconstructing this particular path, i.e., for all $i \in \{1,\ldots,n-1\}$,

$$s_{i+1} \notin \mathsf{wrp}^{\eta}_{A}.\pi_{i+1}.(\Sigma \setminus s') \implies s_i \notin \mathsf{wrp}^{\eta}_{A}.\pi_i.(\Sigma \setminus s') \ ,$$

so with $i = 1$ we are done.
□

Of course, computations which do yield a desired result are also of interest but to prove similar laws one has to argue differently – quantify universally – because predicate transformers demand *all* outcomes to yield those results. Similarly to the previous ones, the names of the following results are inspired by their backward usage. Furthermore, also as done before, when entering a blk-case we refer to the fresh procedure identifier by p' and we identify $\mathsf{blk}(p, \pi_p, \pi_b)$ with $\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p])$ which lets hypotheses apply to renamed components, too.

Lemma 7.4.12 (Backward termination-lemma). For all η, $\boldsymbol{\rho}$, π, A, ψ, s:

$$\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \ \wedge \ (\forall \sigma : \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \sigma : \ \sigma \in \psi \cup A) \\ \Longrightarrow s \in \mathsf{wrp}^{\eta}_{A}.\pi.\psi$$

Proof by structural induction. If the [Skip]-rule applies, then $s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{skip}.\psi$ iff $s \in \psi$ which holds by the premise. For the [Asg]-rules one has $s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{assign}(x, e).\psi$ iff either $\mathcal{E}(e)(s) \in A$ or $\mathcal{E}(e)(s) \in \mathit{Val}$ and $s \in \psi[e/x]$ which also holds by the premise. Now, assume the [Seq3]-rule applies. Then no regular results can be obtained so that in particular, for all ψ,

$$\begin{array}{ll} & \forall \phi, \omega : \ \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1, \pi_2), s\rangle \to \omega : \ \omega \in \phi \cup A \\ \Longrightarrow & \quad \{\text{Operational semantics}\} \\ & \forall \phi, \omega : \ \langle \boldsymbol{\rho}, \pi_1, s\rangle \to \omega : \ \omega \in \phi \cup A \\ \Longrightarrow & \quad \{\text{Structural hypothesis applied to } \pi_1\} \\ & \forall \phi :: \ s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.\phi \\ \Longrightarrow & \quad \{\text{Instantiation}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \Longleftrightarrow & \quad \{\text{Definition}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\psi \ , \end{array}$$

note that universal quantification is antitonic in the range argument. In the [Blk1]-rule one has

$$\begin{array}{ll} & \forall \sigma : \ \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s\rangle \to \sigma : \ \sigma \in \psi \cup A \\ \Longrightarrow & \quad \{\text{Operational semantics}\} \\ & \forall \sigma : \ \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi_b[p'/p], s\rangle \to \sigma : \ \sigma \in \psi \cup A \\ \Longrightarrow & \quad \{\text{Structural hypothesis applied to } \pi_b[p'/p], \\ & \qquad \text{Lemma 7.4.6 allows this}\} \\ & s \in \mathsf{wrp}^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}_{A}.\pi_b[p'/p].\psi \\ \Longleftrightarrow & \quad \{\text{Definition}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]).\psi \\ \Longleftrightarrow & \quad \{\text{Renaming Theorem 7.4.1}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p, \pi_p, \pi_b).\psi \ . \end{array}$$

As the [Cond3]-rule again does not extend our horizon we come to the [Call2]-rule where we note that "ProcUndecl" $\in A$ by the premise. Thus, exploiting $\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A)$,

$$s \in \mathtt{true} \;=\; \top.\psi \;=\; \eta(p).\psi \;=\; \mathsf{wrp}^{\eta}_{A}.\mathsf{call}(p).\psi$$

completes the proof.
□

Here is the expected "backward" version of Lemma 7.4.10 where it again becomes clearer where the name "backward" stems from.

Lemma 7.4.13 (Backward step-lemma). For all η, $\boldsymbol{\rho}$, π, A, ψ, s:

$$\begin{array}{l}\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \;\wedge \\ (\forall \pi', s' : \; \langle \boldsymbol{\rho}, \pi, s\rangle \to \langle \boldsymbol{\rho}, \pi', s'\rangle : \; s' \in \mathsf{wrp}^{\eta}_{A}.\pi'.\psi) \\ \quad \Longrightarrow s \in \mathsf{wrp}^{\eta}_{A}.\pi.\psi\end{array}$$

Proof by structural induction, as usual. We start with the [Seq1]-rule. Here,

$$\begin{array}{ll} & \forall \pi_1', s' : \; \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1, \pi_2), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1', \pi_2), s'\rangle : \\ & \qquad s' \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1', \pi_2).\psi \\ \Longleftrightarrow & \quad \{\mathsf{wrp}^{\eta}_{A} \text{ for sequential composition}\} \\ & \forall \pi_1', s' : \; \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1, \pi_2), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1', \pi_2), s'\rangle : \\ & \qquad s' \in \mathsf{wrp}^{\eta}_{A}.\pi_1'.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \Longrightarrow & \quad \{\text{Operational Semantics}\} \\ & \forall \pi_1', s' : \; \langle \boldsymbol{\rho}, \pi_1, s\rangle \to \langle \boldsymbol{\rho}, \pi_1', s'\rangle : \; s' \in \mathsf{wrp}^{\eta}_{A}.\pi_1'.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \Longrightarrow & \quad \{\text{Structural hypothesis applied to } \pi_1\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \Longleftrightarrow & \quad \{\mathsf{wrp}^{\eta}_{A} \text{ for sequential composition}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\psi \;\; . \end{array}$$

For [Seq2] one has to argue differently, namely

$$\begin{array}{ll} & \forall s' : \; \langle \boldsymbol{\rho}, \mathsf{seq}(\pi_1, \pi_2), s\rangle \to \langle \boldsymbol{\rho}, \pi_2, s'\rangle : \; s' \in \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \\ \Longrightarrow & \quad \{\text{Operational semantics}\} \\ & \forall s' : \; \langle \boldsymbol{\rho}, \pi_1, s\rangle \to s' : \; s' \in \mathsf{wrp}^{\eta}_{A}.\pi_2.\psi \\ \Longrightarrow & \quad \{\text{Lemma 7.4.12}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\pi_1.(\mathsf{wrp}^{\eta}_{A}.\pi_2.\psi) \\ \Longleftrightarrow & \quad \{\mathsf{wrp}^{\eta}_{A} \text{ for sequential composition}\} \\ & s \in \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2).\psi \;\; . \end{array}$$

The claim for the [Blk1]-rule follows from:

$$\begin{array}{ll} & \forall \pi', s' : \; \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi'), s'\rangle : \\ & \qquad s' \in \mathsf{wrp}^{\eta}_{A}.\mathsf{blk}(p, \pi_p, \pi').\psi \\ \Longleftrightarrow & \quad \{\text{Renaming Theorem 7.4.1}\} \end{array}$$

$$
\begin{array}{ll}
& \forall \pi', s' : \ \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi'), s'\rangle : \\
& \qquad s' \in \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p', \pi_p[p'/p], \pi'[p'/p]).\psi \\
\iff & \quad \{\mathsf{wrp}_A^{\eta} \text{ for blocks}\} \\
& \forall \pi', s' : \ \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi_b), s\rangle \to \langle \boldsymbol{\rho}, \mathsf{blk}(p, \pi_p, \pi'), s'\rangle : \\
& \qquad s' \in \mathsf{wrp}_A^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}.\pi'[p'/p].\psi \\
\implies & \quad \{\text{Operational semantics}\} \\
& \forall \pi', s' : \ \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi_b[p'/p], s\rangle \to \\
& \qquad\qquad \langle \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, \pi'[p'/p], s'\rangle : \\
& \qquad s' \in \mathsf{wrp}_A^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}.\pi'[p'/p].\psi \\
\implies & \quad \{\text{Structural hypothesis applied to } \pi_b[p'/p],\} \\
& \quad \text{Lemma 7.4.6 allows this}\} \\
& s \in \mathsf{wrp}_A^{\eta[p' \mapsto \lambda B_{p', \pi_p[p'/p], A, \eta}]}.\pi_b[p'/p].\psi \\
\iff & \quad \{\text{Definition}\} \\
& s \in \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]).\psi \\
\iff & \quad \{\text{Renaming Theorem 7.4.1}\} \\
& s \in \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p, \pi_p, \pi_b).\psi \ .
\end{array}
$$

We skip the [Cond]-rules and for the [While]-rule we have a look at the proof of Lemma 7.4.10, where $\mathsf{wrp}_A^{\eta}.\mathsf{while}(b, \pi) = \mathsf{wrp}_A^{\eta}.\mathsf{if}(b, \mathsf{seq}(\pi, \mathsf{while}(b, \pi)), \mathsf{skip})$ is shown which proves the claim for this case, too. Finally, in the [Call2]-case the premise ensures that $s \in \mathsf{wrp}_A^{\eta}.\rho_{j_p}(p).\psi$ whenever $\langle \boldsymbol{\rho}, \mathsf{call}(p), s\rangle \to \langle \boldsymbol{\rho}, \rho_{j_p}(p), s\rangle$. Exploiting $\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A)$ and $p \in \mathrm{dom}(\boldsymbol{\rho})$ yields

$$s \in \mathsf{wrp}_A^{\eta}.\mathsf{call}(p).\psi \ = \ \eta(p).\psi \ = \ \mathsf{wrp}_A^{\eta}.\rho_{j_p}(p).\psi$$

and we are done.
□

The "backward" counterpart of the forward sequence-lemma 7.4.11 is given below, cf. Lemma 7.4.15, because it does not really fit the stepwise presentation here.

The actual equivalence-proof. For the sake of comprehension the proof is divided into four smaller and handy parts. The first two of them take the laws about steps and termination into account. After the preparations made before the proofs are clearer now and the first fourth reads like this.

Lemma 7.4.14 ($\frac{1}{4}$-equivalence). For all η, $\boldsymbol{\rho}$, π, A: If $\infty \in A$ then

$$\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \quad \implies \quad \mathsf{wrp}_A^{\eta}.\pi \leq \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi \ .$$

Proof. We start with massaging the goal:

$$
\begin{array}{ll}
& \mathsf{wrp}_A^{\eta}.\pi \leq \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi \\
\iff & \quad \{\text{Pointwise definitions of '}\leq\text{' and '}\subseteq\text{'}\} \\
& \forall \psi, s : \ s \in \mathsf{wrp}_A^{\eta}.\pi.\psi : \ s \in \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi.\psi
\end{array}
$$

$$\begin{array}{ll} \iff & \{\text{Trading the range twice}\} \\ & \forall \psi, s: \ s \notin \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi.\psi : \ s \notin \mathsf{wrp}_A^{\eta}.\pi.\psi \ . \end{array}$$

So take some arbitrarily chosen ψ and s such that $s \notin \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi.\psi$. Firstly, we observe that it is *not* the case that $\langle \boldsymbol{\rho}, \pi, s\rangle \to^{\infty}$ because this would entail $s \in \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi.\psi$ as we accept divergence here ($\infty \in A$). Hence, there exist some σ with $\langle \boldsymbol{\rho}, \pi, s\rangle \xrightarrow{+} \sigma$ such that $\sigma \notin \psi \cup A$ and we choose an arbitrary one of those. If σ is a regular result, i.e. $\sigma \in \Sigma$, then $s \notin \mathsf{wrp}_A^{\eta}.\pi.(\Sigma \setminus \sigma)$ by the forward sequence-lemma 7.4.11. As $\sigma \notin \psi$ and $\mathsf{wrp}_A^{\eta}.\pi$ is monotonic, $s \notin \mathsf{wrp}_A^{\eta}.\pi.\psi$ follows. If, otherwise, $\sigma \in \Omega$ then the second claim of Lemma 7.4.11 applies because $\sigma \notin A$. Hence $s \notin \mathsf{wrp}_A^{\eta}.\pi.\mathtt{true}$ and consequently $s \notin \mathsf{wrp}_A^{\eta}.\pi.\psi$ by monotonicity.
□

The second fourth makes heavy use of the "backward" laws about steps and terminations and is the announced counterpart of Lemma 7.4.11.

Lemma 7.4.15 ($\frac{2}{4}$-equivalence). For all η, $\boldsymbol{\rho}$, π, A: If $\infty \notin A$ then

$$\mathsf{bindings}_e(\eta, \boldsymbol{\rho}, A) \implies \mathsf{wrp}_A^{\eta}.\pi \geq \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi \ .$$

Proof. Though the premise $\infty \notin A$ suggest that it suffices to consider finite paths and thus to perform an induction on path lengths it turns out to be much clearer to go another way because the universal quantification over all (finite) paths leaves it unclear which paths to choose resp. how to deal with a variety of finite paths in connection with induction. However, by Theorem 7.1.3 we know that the claim is equivalent to

$$\mathsf{wrp}_A^{\eta}.\pi \geq \mu D.A.(\boldsymbol{\rho}.\pi) \ ,$$

and it is this formulation which is shown by fixpoint induction for μD. Admissibility is clear – D is a lifted function – and so is the base case using the bottom element of the according lattice. So suppose given a function f with

$$\mathsf{wrp}_A^{\eta}.\pi \geq f.A.(\boldsymbol{\rho}.\pi)$$

for all η, $\boldsymbol{\rho}$ and π which satisfy the premise. The induction step is easy, too. Choose some arbitrary ψ and s and observe that

$$\begin{array}{ll} & s \in D.f.A.(\boldsymbol{\rho}, \pi).\psi \\ \iff & \quad \{\text{Definition of } D \text{ resp. an equivalent version}\} \\ & (\forall \sigma : \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \sigma : \ \sigma \in \psi \cup A) \ \wedge \\ & (\forall \pi', s' : \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \langle \boldsymbol{\rho}, \pi', s'\rangle : \ s' \in f.A.(\boldsymbol{\rho}, \pi').\psi) \\ \implies & \quad \{\text{Fixpoint induction hypothesis}\} \\ & (\forall \sigma : \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \sigma : \ \sigma \in \psi \cup A) \ \wedge \\ & (\forall \pi', s' : \ \langle \boldsymbol{\rho}, \pi, s\rangle \to \langle \boldsymbol{\rho}, \pi', s'\rangle : \ s' \in \mathsf{wrp}_A^{\eta}.\pi'.\psi) \\ \implies & \quad \{\text{Backward Lemmas 7.4.12 and 7.4.13}\} \\ & s \in \mathsf{wrp}_A^{\eta}.\pi.\psi \ , \end{array}$$

such that $\mathsf{wrp}^{\eta}_{A}.\pi \geq D.f.A.(\boldsymbol{\rho}.\pi)$ follows immediately.[15]
□

This completes the first half of the overall claim. For the second half we benefit from the abstractions, the algebraic laws, concerning the dictionary based wrp-transformers. Apart from the procedure mechanism the derived laws look quite similar to the definition of the environment based wrp-transformers. The missing parts, the procedure concept, can nicely be managed under appropriate assumptions concerning the relation between syntactical and semantical bindings and, luckily, we already know those relations: They are kept in the $\mathsf{bindings}_{\lambda}$ predicates.

Lemma 7.4.16 ($\frac{3}{4}$-equivalence). For all η, $\boldsymbol{\rho}$, π, A: If $\infty \notin A$ then

$$\mathsf{bindings}_{\mu}(\eta, \boldsymbol{\rho}, A) \quad \Longrightarrow \quad \mathsf{wrp}^{\eta}_{A}.\pi \leq \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\pi \ .$$

Proof. A structural induction is performed, i.e. we assume Lemma 7.4.16 to hold for the components of which the program in question consists. The base cases are obvious because

$$\mathsf{wrp}^{\eta}_{A}.\mathsf{skip} \ = \ Id \ = \ \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{skip}$$

and

$$\mathsf{wrp}^{\eta}_{A}.\mathsf{assign}(x,e) \ = \ (x :=_{A} e) \ = \ \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{assign}(x,e)$$

by the definitions resp. the algebraic laws derived before. For sequential compositions one has

$$\begin{array}{ll} & \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1,\pi_2) \\ = & \quad \{\text{Definition}\} \\ & \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}^{\eta}_{A}.\pi_2 \\ \leq & \quad \{\text{Structural hypothesis applied to the components}\} \\ & \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\pi_1 \ ; \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\pi_2 \\ \leq & \quad \{\text{Algebraic law}\} \\ & \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{seq}(\pi_1,\pi_2) \ . \end{array}$$

For conditional-commands one argues analogously, i.e. one exploits the structural hypothesis for the components. For loops we start with

$$\begin{array}{ll} & \mathcal{W}_{b,\mathsf{wrp}^{\eta}_{A}.\pi}(\mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{while}(b,\pi)) \\ = & \quad \{\text{Definition}\} \\ & (\mathsf{wrp}^{\eta}_{A}.\pi \ ; \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{while}(b,\pi)) \lhd b/A \rhd Id \\ \leq & \quad \{\text{Structural hypothesis applied to } \pi\} \\ & (\mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\pi \ ; \mathsf{wrp}^{\boldsymbol{\rho}}_{A}.\mathsf{while}(b,\pi)) \lhd b/A \rhd Id \\ \leq & \quad \{\text{Algebraic law for sequential composition}\} \end{array}$$

[15] We like to mention that both conjuncts have to be considered because we show the claim for all π and all transitions at once, including those for which only one of the conjuncts does apply such that we could assure $s \in \mathtt{true}$ at most if it was omitted.

$$
\begin{array}{ll}
& \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{seq}(\pi, \mathsf{while}(b,\pi)) \lhd b/A \rhd Id \\
= & \quad \{\text{Algebraic law for conditional and skip}\} \\
& \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{if}(b, \mathsf{seq}(\pi, \mathsf{while}(b,\pi)), \mathsf{skip}) \\
= & \quad \{\text{Algebraic law for loops}\} \\
& \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi) \ ,
\end{array}
$$

such that $\mu\mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi} \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi)$ follows from the induction rule and consequently $\mathsf{wrp}^{\eta}_A.\mathsf{while}(b,\pi) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi)$ by definition.[16] Blocks can be managed by exploiting the $\mathsf{bindings}_\mu$-predicate. Suppose $\pi = \mathsf{blk}(p, \pi_p, \pi_b)$ and assume given an arbitrary procedure identifier p' which is fresh w.r.t. $\boldsymbol{\rho}$ and π. Then, by the Renaming Theorems 7.4.1 and 7.4.2, showing

$$\mathsf{wrp}^{\eta}_A.\mathsf{blk}(p, \pi_p, \pi_b) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p, \pi_p, \pi_b)$$

is equivalent to showing

$$\mathsf{wrp}^{\eta}_A.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]) \ ,$$

and it is this inequality that will be established. As before, it has the particular advantage that it allows to apply the structural hypothesis to renamed components.

$$
\begin{array}{ll}
& \mathsf{wrp}^{\eta}_A.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]) \\
= & \quad \{\text{Definition}\} \\
& \mathsf{wrp}^{\eta[p' \mapsto \mu B_{p', \pi_p[p'/p], A, \eta}]}_A.\pi_b[p'/p] \\
\leq & \quad \{\text{Structural hypothesis applied to } \pi_b[p'/p], \text{ see below}\} \\
& \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p' \mapsto \pi_p[p'/p]\}}_A.\pi_b[p'/p] \\
\leq & \quad \{\text{Algebraic law}\} \\
& \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p, \pi_p, \pi_b) \\
= & \quad \{\text{Renaming Theorem 7.4.2}\} \\
& \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{blk}(p', \pi_p[p'/p], \pi_b[p'/p]) \ .
\end{array}
$$

It remains to justify why application of the structural hypothesis is admissible in the second step. It would certainly be clear if

$$\mathsf{bindings}_\mu(\eta[p' \mapsto \mu B_{p', \pi_p[p'/p], A, \eta}], \boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\}, A) \ ,$$

and indeed this is the case. We argue similar to Lemma 7.4.6 and firstly observe that $\boldsymbol{\rho}\cdot\{p' \mapsto \pi_p[p'/p]\}$ is distinguished because $\boldsymbol{\rho}$ is distinguished and $p' \notin \mathrm{dom}(\boldsymbol{\rho})$. Now suppose given an arbitrary $q \in \mathit{Procname}$. The case $q \notin \mathrm{dom}(\boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\})$ is analogous to Lemma 7.4.6, a precise look at the premise suffices, so assume $q \in \mathrm{dom}(\boldsymbol{\rho} \cdot \{p' \mapsto \pi_p[p'/p]\})$, i.e. either $q \in \mathrm{dom}(\boldsymbol{\rho})$ and $q \neq p'$ or $q \notin \mathrm{dom}(\boldsymbol{\rho})$ and $q = p'$. In the first case it follows from the assumptions about $\mathsf{bindings}_\mu(\eta, \boldsymbol{\rho}, A)$ that

[16] If the reader believes in all algebraic laws, even those from which we said they will not be needed, this case might have been handled easier: If $\mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi)$ is any fixpoint of $\mathcal{W}_{b,\mathsf{wrp}^{\boldsymbol{\rho}}_A.\pi}$ then $\mathsf{wrp}^{\eta}_A.\mathsf{while}(b,\pi) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{while}(b,\pi)$ because $\mathsf{wrp}^{\eta}_A.\mathsf{while}(b,\pi)$ is the least fixpoint of $\mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi}$ and the bodies of both functions are related by '$\leq$' by the hypothesis.

$$\eta[p' \mapsto \mu B_{p',\pi_p[p'/p],A,\eta}](q) = \eta(q) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\rho_{j_q}(q) \ ,$$

and here, in the end, Lemma 7.4.4, i.e. the fact that $\rho_{j_q}(q)$ does not call p' as p' is fresh w.r.t. $\boldsymbol{\rho}$, guarantees that

$$\eta[p' \mapsto \mu B_{p,\pi_p[p'/p],A,\eta}](q) \leq \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\rho_{j_q}(q)$$

as required. More interesting is the second case where it remains to show that

$$(*) \quad \mu B_{p',\pi_p[p'/p],A,\eta} \leq \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p]$$

which is screaming for the induction rule. Thus, we start our calculations with

$$\begin{array}{ll} & B_{p',\pi_p[p'/p],A,\eta}(\mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p]) \\ = & \quad \{\text{Definition of } B\} \\ & \mathsf{wrp}^{\eta[p'\mapsto\mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p]]}_A.\pi_p[p'/p] \ , \end{array}$$

and it would follow from the structural hypothesis applied to $\pi_p[p'/p]$ that the last line is less than or equal to the right hand side of $(*)$ if

$$\mathsf{bindings}_\mu(\eta[p' \mapsto \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p]], \boldsymbol{\rho}\cdot\{p' \mapsto \pi_p[p'/p]\}, A) \ .$$

After similar arguments as just made before it remains to accept that

$$\mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p] \leq \mathsf{wrp}^{\boldsymbol{\rho}\cdot\{p'\mapsto\pi_p[p'/p]\}}_A.\pi_p[p'/p]$$

which is not worth mentioning. Altogether, this proves the blk-case. Finally, procedure calls demand two cases, namely

$$\begin{array}{ll} & \mathsf{wrp}^{\eta}_A.\mathsf{call}(p) \\ = & \quad \{\text{Definition}\} \\ & \eta(p) \\ \leq & \quad \{\text{See } \mathsf{bindings}_\mu\} \\ & \mathsf{wrp}^{\boldsymbol{\rho}}_A.\rho_{j_p}(p) \\ = & \quad \{\text{Algebraic law}\} \\ & \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{call}(p) \ , \end{array}$$

where $\eta(p) \leq \mathsf{wrp}^{\boldsymbol{\rho}}_A.\rho_{j_p}(p)$ is known literally from $\mathsf{bindings}_\mu(\eta, \boldsymbol{\rho}, A)$ if $p \in \mathrm{dom}(\boldsymbol{\rho})$. Similarly, in the case $p \notin \mathrm{dom}(\boldsymbol{\rho})$ one has

$$\begin{array}{ll} & \mathsf{wrp}^{\eta}_A.\mathsf{call}(p) \\ = & \quad \{\text{Definition}\} \\ & \eta(p) \\ = & \quad \{\text{See } \mathsf{bindings}_\mu\} \\ & \begin{cases} \top & : \ \text{“ProcUndecl”} \in A \\ \bot & : \ \text{“ProcUndecl”} \notin A \end{cases} \\ = & \quad \{\text{Algebraic law}\} \\ & \mathsf{wrp}^{\boldsymbol{\rho}}_A.\mathsf{call}(p) \end{array}$$

which completes the proof.

□

By symmetry the last fourth of the equivalence-proof looks similar.

Lemma 7.4.17 ($\frac{4}{4}$-equivalence). For all η, $\boldsymbol{\rho}$, π, A: If $\infty \in A$ then

$$\mathsf{bindings}_\nu(\eta, \boldsymbol{\rho}, A) \quad \Longrightarrow \quad \mathsf{wrp}_A^\eta.\pi \geq \mathsf{wrp}_A^{\boldsymbol{\rho}}.\pi \ .$$

Proof. Not only the lemma itself reads quite similar, the proof does either. A structural induction is performed where the base cases are exactly the same as above and in the induction steps one uses the hypothesis, the algebraic laws and the definitions time and again. Note that loops and blocks are defined via greatest fixpoints such that the induction rule for greatest fixpoint applies and observe that the premise $\mathsf{bindings}_\nu$ allows an analogous proceeding, too.
□

This $\frac{4}{4}$-proof completes the preparations for the main issue of this section. Each of the four lemmas above handles one inequality w.r.t. $\infty \in A$ or not, and this under certain assumptions concerning the bindings. Luckily, these assumptions hold for the "interesting cases", the mappings that are the adequate ones if both semantics-presentations are to be compared: Choose the initial environment η_{initial} and the empty procedure dictionary $\emptyset$, cf. Lemmas 7.4.5 and 7.4.7.

Theorem 7.4.3 (Equivalence theorem). For all π, A:

$$\mathsf{wrp}_A^{\eta_{\text{initial}}}.\pi \ = \ \mathsf{wrp}_A^{\emptyset}.\pi \ .$$

□

7.5 Remarks

The benefits of this chapter are threefold and worth some comments.

The fixpoint-characterization of wrp-transformers presented in Sect. 7.1 is somewhat fundamental because of its plenty of applications. Rolling and unrolling the fixpoints of the explicitly given function D as defined by (7.4) on p. 89 allows to execute a program in question symbolically and also consistently what the semantics concerns (if an underlying operational semantics is present at all). This seems not to be very exciting at first glance but it is remarkable that each obtained result directly corresponds to a relative correctness property. Executing machine programs symbolically might be well known and even of independent interest but typically there is nothing beyond the actual execution. Moreover, for verification purposes it turns out to be at least helpful to have an explicitly given function like D at hand. If a structural induction does not suffice or apply one is in need of alternative proof-techniques. As recursion is expressed by means of fixpoints one typically tries to make use of simpler fixpoint rules, such as the induction rule or the transfer lemma, but it might well be the case that these techniques do not apply because either, roughly speaking, the fixpoints are not isolated or on the wrong side of an inequality sign. Brute force, i.e. fixpoint induction in this case, is thus the final attempt and here function D typically plays the role of the function to induce on, see also Sect. 8.2.

We like to remark that Sect. 7.1 is heavily inspired by Chap. 9 of [31] where similar results are given for the simplified world of wp and wlp in which the presence of errors is disregarded. The underlying relational semantics is marginally different from our '$\rightsquigarrow$'. It is (more or less verbally) defined as an extension of the *reflexive*-transitive closure and, thus, some simple but fundamental properties, like our Lemma 7.1.1, differ (and, moreover, cannot be proved due to lack of formalism). Consequently some of [31]'s arguments change, too, and seem less intuitive.

In some sense, similar considerations can be found in the very worth reading [19]. A so-called computation calculus – i.e. a predicate algebra in the sense of [18] with an additional composition operator satisfying certain axioms – is introduced which is intended to bridge the formalization gap between abstract programming formalisms and the operational interpretations they have been designed for. The wp- and wlp-transformers are defined in the calculus and nicely shown to be equal to some least resp. greatest fixpoints of very intuitive tail-recursive functions. However, those results can only sharpen the intuition because, in contrast to our exposition, there is no underlying operational semantics and more and more powerful axioms have to be included in order to prove the claim at all. As usual, the presence of finite errors is omitted for simplicity.

Some final remarks concerning fixpoints. It might well be the case that least fixpoints are more prominent in computer science because syntax, semantics, data-types etc. are often defined inductively by means of certain, say, rules and typically one is interested in finite application of these rules in order to obtain a result. If, additionally, these rules enjoy some distribution properties finite application indeed coincides with the least fixpoint of a corresponding function or functional. In this sense, a *proof by induction* corresponds to showing that the least fixpoint enjoys a certain property. Thus, such a proof has a very constructive flavor as one takes nothing special for granted and only believes what can be demonstrated to hold, i.e. what is guaranteed to hold *at least*. Typically, properties concerning program behavior are shown by induction because a program should exhibit no behavior which is not documented in the semantics. On the other hand, showing a property to hold for greatest fixpoints has a quite different meaning. Here, everything may be assumed as long as it cannot be shown to violate any facts; one tries to preserve *at most* all valid attributes. A proof-technique like this is often used to show that a specification enjoys certain properties because a specification should not forbid any behavior unless explicitly stated. In category theory this is called a *coinduction* and, in fact, a coinduction is nothing else but an application of certain rules concerning greatest fixpoints, see, e.g., [45] for a discussion.[17] In [64], for instance, this proceeding has been used for some control flow analysis and we advise to have a look at this monograph for some more intuitive explanations. To resume, in general both, least and greatest fixpoints, are natural from their

[17] Let us just mention that one can define a functional, say F, on relations in such ways that its greatest fixpoint νF equals the greatest bisimulation $\sim$. Showing that two transition systems simulate another, thus, amounts to showing a property concerning νF to hold. In this situation a coinduction is exactly an application of the induction rule for greatest fixpoints.

very own perspective. In our wrp-parlance this can be intuitively illustrated as follows. The least fixpoint successively collects – starting with the empty set, i.e. `false` – all states which give rise to computations yielding some desired outcomes. The greatest fixpoint strategy on the other side starts with the universe of states, i.e. `true`, and successively strikes out all states from which a computation might show a certain behavior which is not tolerated. That accepting divergence comes along with the greatest fixpoint is thus due to the fact that "infinity" cannot be reached in finitely many steps but remains in the solution if violating states are canceled one after another.

The assembly language presented in Sect. 7.3 is also worth some comments. Recall that the language is flat, i.e. without any inner structure, has a subroutine concept and is assumed to run on a finite machine. For translation verification purposes (like Sect. 8.2 of the present thesis) typically simpler assembly languages are considered. Most of them are not equipped with procedures, beyond that some have a tree-structure (e.g. [65]), and even if the language is rather realistic in this sense the presence of finite errors is ignored (e.g. [47]). Therefore, to quote ourselves once more, Sect. 7.3 presents a language which is *almost* an abstract view on an existing assembly language and can, thus, be kept for a stepping stone on the way to actual machine code. The appendix discusses this claim at length.

On the other side, the high-level language presented in Sect. 7.4 was mainly introduced in order to show what a wrp-semantics looks like and to have a more concrete semantics at hand to work with in the sequel. Nevertheless, it is nice to see that the denotational wrp-semantics harmonizes with the better known wp- or wlp-semantics well. Its derivation, i.e. the above equivalence-proof, was a harder exercise and might also be of independent interest. Noteworthy in this regard is the fact that it was performed without use of strictness resp. continuity. Typically, these properties are needed in order to match the intuition that least fixpoints coincide with finite iterations. We evaded strictness resp. continuity by using the fixpoint characterization of wrp instead. It is furthermore worth mentioning that the recipe presented here promises to generalize to even more complicated languages. To model the semantics of formal procedure parameters it is unavoidable to store old bindings, this is why we used stacks of dictionaries which allow to refer to lower entries and to forget about later declarations. A similar proceeding might also allow the usage of local variables: As done for procedure identifiers a stack of local variables should be taken into account. In the very end, a proceeding like this amounts to implementing an algebraic counterpart of a so-called activation record (see the classical textbooks) and the exposition presented here is thought to guide through more difficult tasks.

8. "Applications"

It remains is to show that the proposed wrp-transformers keep the promise to facilitate proving programs and particularly translations correct. As mentioned in Chap. 2 the classic notions of preservation of partial and preservation of total correctness can be kept for not adequate for realistic programs running on real machines; let us recall why. On the one hand, code optimizations forbid preservation of partial correctness as they typically expect total correct programs for an input and mostly disregard the presence of errors. But, on the other hand, limited machine resources forbid total correctness and consequently preservation thereof because a program running on a finite machine may stop irregularly, i.e. abort, propagating a finite error though the source program was proved to deliver a regular result.

Now, it is time to return to the roots in the sense that the examples from the beginning are revisited. As some more concrete languages and their wrp-semantics are at hand they can be discussed in a formal manner and precise requirements can be expressed which are needed in order to guarantee preservation of relative correctness. To be more specific, the high-level language from Sect. 7.4 is acting as a source language of a translation. But before an actual compilation we study the mentioned optimizations formally and prove them correct w.r.t. appropriate and well defined premises. Justifications like these are rare in the sense that, typically, the correctness and admissibility of those strategies are given at best verbally in common text-books like [59]. Afterwards, programs of the source language are translated to the flat, unstructured assembly code of Sect. 7.3. Again, we focus on the translation of control structures, i.e. the linearization of loops, conditionals and procedures by jumps. The translation is shown to preserve relative correctness in the sense that, colloquially speaking, partial and total correctness is preserved unless the executing machine is just too small to stack all needed subroutine-return-addresses.[1] This result is of great value as typically the translation of pure WHILE-languages without procedures – running on idealized, infinite machines – is considered where in general preservation of total correctness is the notion of interest. Adding procedures, in particular nested procedures without naming restrictions, assuming the translated programs to run on finite machines and – most of all – formally proving correct the translation in the sense of preservation of partial correctness resp. a variation thereof is a harder task which has been neglected so far.

[1] The essence of half of this proof is already published in [63], see Sect. 8.2.

8.1 Justifying Code Optimizations

Roughly speaking, optimizing a program means to translate a given program to another program of the same language, which is the high-level language of Sect. 7.4 in this case, in order to increase the performance (the execution speed) or to decrease the needed resources (the needed memory). Of course, the outcomes of both programs should remain sensibly related. Thus, we are interested in correct implementations of the given program to be optimized, in refinements and, to be precise, in correct translations in the sense of preservation of relative correctness. What follows is a formal discussion of simplified versions of most of the examples given in Chap. 2 added by some appropriate, needed and helpful definitions and auxiliary results. For the sake of readability we make use of a modified syntax of the high-level language. The reader should be quite familiar with syntax and semantics now and is thus requested to accept the notations $x := e$, $\pi\ ;\ \pi'$, $\mathsf{if}\ b\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi}$ and $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$ standing for $\mathsf{assign}(x, e)$, $\mathsf{seq}(\pi, \pi')$, $\mathsf{if}(b, \pi, \pi')$ and $\mathsf{while}(b, \pi)$ respectively. We will be rather careful to distinguish syntactical from semantical operators by using additional parentheses whenever this is appropriate. Furthermore, we take the wrp-transformers as defined in Def. 7.4.2 for the underlying semantics of programs. As the below examples do not make use of procedures we omit the environment argument.

8.1.1 Dead code elimination

Consider the following two little programs:

$$\pi_1 \ = \ x := e\ ;\ x := f\ ;\ \pi \quad \text{and} \quad \pi_2 \ = \ x := f\ ;\ \pi\ ,$$

where π is assumed to be an arbitrary program. If expression f is independent of variable x it is intuitively safe to remove the assignment $x := e$ from π_1, yielding π_2, as the value of x is immediately over-written; the assignment $x := e$ is waste in this sense. Thus, the claim here reads

$$\mathsf{wrp}_A.\pi_1 \ \leq \mathsf{wrp}_A.\pi_2 \tag{8.1}$$

for an appropriate set A of outcomes to be accepted.

To prove the claim, and to find the searched set A, we firstly have to integrate the sentence "f is independent of x" into our vocabulary. Intuitively this means that expression f's value does not depend on variable x's value or, in other words, whatever x might be assigned to by a state s, the value of f remains the same. We like to define this intuition as follows. Expression f is said to be *independent* of variable x iff

$$\mathcal{E}(f)(s) \ = \ \mathcal{E}(f)(s\{x \mapsto v\})$$

for all values $v \in \mathit{Val}$ and states $s \in \Sigma$. A simple consequence is the fact that

$$\mathcal{E}(f)(s) \ = \ \mathcal{E}(f)(s\{x \mapsto \mathcal{E}(e)(s)\})$$

for all expressions e and states $s \in \mathsf{def}(e)$ if f is independent of x. Yet some other consequences are the following. Recall the set $\mathsf{in}_O(f)$ of states which yield an outcome contained in $O \subseteq \Sigma \cup \Omega$, see (7.9). If f is independent of x then

$$
\begin{array}{ll}
 & s \in \mathsf{in}_O(f) \\
\iff & \quad \{\text{Definition of } \mathsf{in}_O(f)\} \\
 & \mathcal{E}(f)(s) \in O \\
\iff & \quad \{\text{Assume } s \in \mathsf{def}(e),\ f \text{ is independent of } x\} \\
 & \mathcal{E}(f)(s\{x \mapsto \mathcal{E}(e)(s)\}) \in O \\
\iff & \quad \{\text{Definition of } \mathsf{in}_O(f)\} \\
 & s\{x \mapsto \mathcal{E}(e)(s)\} \in \mathsf{in}_O(f) \\
\iff & \quad \{\text{Definition of substitution}\} \\
 & s \in \mathsf{in}_O(f)[e/x]\ .
\end{array}
$$

Furthermore it is intuitively clear that, and this is the key to eliminate dead or redundant code, a substitution can be cancelled if it has no effect in the following sense. By definition of what we dared to call a "syntactical substitution" of e for x in f, i.e. $f\langle e/x\rangle$, cf. (7.7), one has

$$
\begin{array}{ll}
 & \mathcal{E}(f\langle e/x\rangle)(s) \\
= & \quad \{\text{Definition}\} \\
 & \mathcal{E}(f)(s\{x \mapsto \mathcal{E}(e)(s)\}) \\
= & \quad \{\text{Assume } s \in \mathsf{def}(e) \text{ and } f \text{ is independent of } x\} \\
 & \mathcal{E}(f)(s)\ .
\end{array}
$$

The former observation and a combination of the latter with (7.8) result in

Lemma 8.1.1 (Cancel substitution). If f is independent of x, then

$$\mathsf{def}(e) \cap \phi[f/x][e/x] \;=\; \mathsf{def}(e) \cap \phi[f/x]$$

for all ϕ, and

$$\mathsf{def}(e) \cap \mathsf{in}_O(f) \;=\; \mathsf{def}(e) \cap \mathsf{in}_O(f)[e/x]$$

for all $O \subseteq \Sigma \cup \Omega$.
□

To approach the goal (8.1) it is helpful to notice that $\pi_1 = x := e\ ;\ \pi_2$ which allows, under the assumption that f is independent of x, to calculate as follows: For arbitrary ψ and A,

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.\pi_1.\psi \\
= & \quad \{\text{Semantics of } \pi_1\} \\
 & (x :=_A e)(\mathsf{wrp}_A.\pi_2.\psi) \\
= & \quad \{\text{Definition of } (x :=_A e)\} \\
 & \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap (\mathsf{wrp}_A.\pi_2.\psi)[e/x]) \\
= & \quad \{\text{Semantics of } \pi_2\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap (\ \mathsf{in}_A(f) \cup (\mathsf{def}(f) \cap (\mathsf{wrp}_A.\pi.\psi)[f/x])\ )[e/x]) \\
= & \qquad \{\text{Distribute the latter substitution}\} \\
 & \mathsf{in}_A(e) \cup \\
 & (\mathsf{def}(e) \cap (\ \mathsf{in}_A(f)[e/x] \cup (\mathsf{def}(f)[e/x] \cap (\mathsf{wrp}_A.\pi.\psi)[f/x][e/x])\ )) \\
= & \qquad \{\text{Cancel some substitutions with Lemma 8.1.1}\} \\
 & \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap (\ \mathsf{in}_A(f) \cup (\mathsf{def}(f) \cap (\mathsf{wrp}_A.\pi.\psi)[f/x])\ )) \\
= & \qquad \{\text{Semantics of } \pi_2\} \\
 & \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap \mathsf{wrp}_A.\pi_2.\psi)\ .
\end{array}
$$

Thus, (8.1) obviously follows if $\mathsf{in}_A(e) = \mathtt{false}$, i.e. if e does not evaluate to an accepted outcome for any initial state, and indeed, in the absence of further knowledge about the involved expressions e and f and the remainder π this is the only safe statement one can make. Intuitively this means that the translation of π_1 to π_2 is permissible if none of the failures potentially produced by e belongs to the accepted failures in A because otherwise this particular failure would make π_1 relatively correct w.r.t. A whereas the optimized program π_2 might produce just any outcome, irrespective of any specification. This is in particular the case if A does not contain any arithmetic error,[2] i.e. none of the errors produced by arithmetic expressions. For a more far-reaching conclusion one would need more specific knowledge about e. For example, one might accept something like a "DivByZero" if e does not contain a division. However, letting

$$\mathsf{Error}(e) \stackrel{\text{def}}{=} \{\omega \in \Omega \mid \exists s \in \Sigma :: \ \mathcal{E}(e)(s) = \omega\}$$

denote the set of potential erroneous outcomes, arithmetic errors, of e we can formulate the obtained result as follows.[3]

Theorem 8.1.1 (Correctness of dead code elimination). If f is independent of x and if $\mathsf{Error}(e) \cap A = \emptyset$, then

$$\mathsf{wrp}_A.x := e\ ;\ x := f\ ;\ \pi\ \ \leq\ \ \mathsf{wrp}_A.x := f\ ;\ \pi\ .$$

□

It is interesting to discuss also the border cases for this example. In the PTC-case one has $A = \emptyset$, so $\mathsf{in}_A(e)$ equals $\mathtt{false}$ for trivial reasons. Thus π_2 indeed implements π_1 w.r.t. PTC. In the PPC case, on the other hand we have to choose $A = \Omega$. Then $\mathsf{in}_A(e)$ might be valid for some states if evaluation of e might fail. Thus, the transformation might be invalid in the sense of PPC, depending on the shape of e. Thus, the formal framework confirms our informal reasoning from the beginning, Sect. 2.2, and the wrp-transformers allow to reason about questions like these, just as promised.

[2] Formally, an outcome ω is an *arithmetic error* if there exists an expression e and a state s such that $\mathcal{E}(e)(s) = \omega$. In this sense e produces an arithmetic error if $\mathsf{in}_\Omega(e) \neq \mathtt{false}$.

[3] In practice one would rather claim that A does not contain any arithmetic error.

8.1.2 Code motion

Again, we consider a slightly simplified version of the code motion example presented in Sect. 2.2:

$$\pi_1 \;=\; \mathsf{if}\ b\ \mathsf{then}\ x := e\ ;\ y := g\ \mathsf{else}\ x := f\ ;\ y := g\ \mathsf{fi}$$

and

$$\pi_2 \;=\; y := g\ ;\ \mathsf{if}\ b\ \mathsf{then}\ x := e\ \mathsf{else}\ x := f\ \mathsf{fi}\ ,$$

where we assume x and y to be distinct, i.e. $x \neq y$ (otherwise the former assignments may be cancelled if desired, see above).

Independence is again essential and we start with some more rules concerning this notion. Under certain assumptions the order of assignments can be changed, namely if each of the expressions involved is independent of each of the others variables. Before showing this one observes that

$$\begin{array}{ll}
 & s \in \psi[g/y][e/x] \\
\iff & \quad \{\text{Definition of substitution...}\} \\
 & s\{x \mapsto \mathcal{E}(e)(s)\} \in \psi[g/y] \\
\iff & \quad \{\text{...and once again}\} \\
 & s\{x \mapsto \mathcal{E}(e)(s)\}\{y \mapsto \mathcal{E}(g)(s\{x \mapsto \mathcal{E}(e)(s)\})\} \in \psi \\
\iff & \quad \{\text{Assume } g \text{ is independent of } x\} \\
 & s\{x \mapsto \mathcal{E}(e)(s)\}\{y \mapsto \mathcal{E}(g)(s)\} \in \psi \\
\iff & \quad \{\text{Rearrange variations, } x \neq y\} \\
 & s\{y \mapsto \mathcal{E}(g)(s)\}\{x \mapsto \mathcal{E}(e)(s)\} \in \psi \\
\iff & \quad \{\text{Assume } e \text{ is independent of } y\} \\
 & s\{y \mapsto \mathcal{E}(g)(s)\}\{x \mapsto \mathcal{E}(e)(s\{y \mapsto \mathcal{E}(g)(s)\})\} \in \psi \\
\iff & \quad \{\text{Reverse first two steps}\} \\
 & s \in \psi[e/x][g/y]\ ,
\end{array}$$

which proves

Lemma 8.1.2 (Rearrange substitutions). If $x \neq y$, g is independent of x and e is independent of y, then

$$\mathsf{def}(g) \cap \mathsf{def}(e) \cap \psi[g/y][e/x] \;=\; \mathsf{def}(g) \cap \mathsf{def}(e) \cap \psi[e/x][g/y]\ .$$

□

Furthermore, taking the premises of Lemma 8.1.2 for granted, the cancelation of substitutions with Lemma 8.1.1 yields, e.g.,

$$\begin{array}{lll}
\mathsf{def}(g) \cap \mathsf{in}_A(e)[g/y] & = & \mathsf{def}(g) \cap \mathsf{in}_A(e)\ , \\
\mathsf{def}(e) \cap \mathsf{in}_A(g)[e/x] & \subseteq & \mathsf{in}_A(g)\ , \text{ and} \\
\mathsf{def}(e) \cap \mathsf{def}(g)[e/x] & \subseteq & \mathsf{def}(g)\ ,
\end{array}$$

It follows that

$$\begin{array}{ll} & \mathsf{def}(e) \cap \mathsf{def}(g)[e/x] \\ = & \quad \{\text{A consequence of the latter observation above}\} \\ & \mathsf{def}(e) \cap \mathsf{def}(g)[e/x] \cap \mathsf{def}(g) \\ \subseteq & \quad \{\text{Cancel substitution with Lemma 8.1.1}\} \\ & \mathsf{def}(e)[g/y] \ , \end{array}$$

such that the combination of the latter two inclusions and Lemma 8.1.2 ensures

$$\mathsf{def}(e) \cap \mathsf{def}(g)[e/x] \cap \psi[g/y][e/x] \ \subseteq \ \mathsf{def}(e)[g/y] \cap \psi[e/x][g/y] \ .$$

Let us finally assume that g always yields a regular result or an accepted irregular outcome whenever e produces an accepted irregular outcome, i.e. $\mathsf{in}_A(e) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$. Then all preparations are made to see that

$$\begin{array}{l} \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap (\ \mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap \psi[g/y])\ )[e/x]) \\ \quad \subseteq \mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap (\ \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap \psi[e/x])\ )[g/y]) \ . \end{array}$$

Unrolling the definitions of assignments and sequential composition, this proves the first essential claim.

Lemma 8.1.3 (Rearrange assignments). If $x \neq y$, g is independent of x, e is independent of y and if $\mathsf{in}_A(e) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$, then

$$\mathsf{wrp}_A.(x := e \ ; \ y := g) \ \leq \ \mathsf{wrp}_A.(y := g \ ; \ x := e) \ .$$

□

In some sense the second assignment has jumped to the left and it remains to show that it can also jump out of the conditional. From Lemma 7.2.1 in Sect. 7.2 we know that sequential composition from the right distributes over conditionals. In rare cases sequential composition distributes over conditionals also from the left so let us collect the ingredients to prove this. Firstly, it holds that

$$(b = \mathsf{tt}) \cap \mathsf{def}(g) \ \subseteq \ (b = \mathsf{tt})[g/y]$$

if b is independent of y, see Lemma 8.1.1. Hence, for all predicates ϕ and ϕ',

$$\begin{array}{l} (b = \mathsf{tt}) \cap (\mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap \phi[g/y])) \\ \quad \subseteq \mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap (\ \mathsf{in}_A(b)[g/y] \cup \\ \qquad\qquad\qquad\qquad ((b = \mathsf{tt})[g/y] \cap \phi[g/y]) \cup \\ \qquad\qquad\qquad\qquad ((b = \mathsf{ff})[g/y] \cap \phi'[g/y])\ )) \ , \end{array}$$

and analogous statements hold for $(b = \mathsf{ff})$. Assuming that g always yields a regular result or an accepted irregular outcome whenever b produces an accepted irregular outcome, i.e. $\mathsf{in}_A(b) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$, one obtains

$$
\begin{array}{ll}
 & \mathsf{in}_A(b) \cup \\
 & ((b = \mathsf{tt}) \cap (\mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap \phi[g/y]))) \cup \\
 & ((b = \mathsf{ff}) \cap (\mathsf{in}_A(g) \cup (\mathsf{def}(g) \cap \phi'[g/y]))) \\
\subseteq & \quad \{\text{Auxiliary results above}\} \\
 & \mathsf{in}_A(g) \cup \\
 & (\mathsf{def}(g) \cap (\ \mathsf{in}_A(b)[g/y] \cup \\
 & \qquad\qquad ((b = \mathsf{tt})[g/y] \cap \phi[g/y]) \cup \\
 & \qquad\qquad ((b = \mathsf{ff})[g/y] \cap \phi'[g/y])\ ))\ ,
\end{array}
$$

if b is independent of y, note that again $\mathsf{def}(g) \cap \mathsf{in}_A(b)[g/y] = \mathsf{def}(g) \cap \mathsf{in}_A(b)$. Now unroll the definitions and see that sequential composition from the right may indeed distribute over conditionals in the following ways.

Lemma 8.1.4 (Distribute conditional and independence). If b is independent of y and $\mathsf{in}_A(b) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$, then, for all $P, Q \in PTrans$,

$$
\begin{array}{l}
((y :=_A g)\ ;\ P) \lhd b/A \rhd ((y :=_A g)\ ;\ Q) \\
\quad \leq (y :=_A g)\ ;\ (P \lhd b/A \rhd Q)\ .
\end{array}
$$

□

This completes the preparations to consider code motion in more detail. Under the assumptions that b, e and f are independent of y, g is independent of x and that $\mathsf{in}_A(b) \cup \mathsf{in}_A(e) \cup \mathsf{in}_A(f) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$ we argue as follows:

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.(\mathsf{if}\ b\ \mathsf{then}\ x := e\ ;\ y := g\ \mathsf{else}\ x := f\ ;\ y := g\ \mathsf{fi}) \\
= & \quad \{\text{Semantics of conditional}\} \\
 & \mathsf{wrp}_A.(x := e\ ;\ y := g) \lhd b/A \rhd \mathsf{wrp}_A.(x := f\ ;\ y := g) \\
\leq & \quad \{\text{Rearrange assignments with Lemma 8.1.3,} \\
 & \quad\ \text{conditional is monotonic in its branches}\} \\
 & \mathsf{wrp}_A.(y := g\ ;\ x := e) \lhd b/A \rhd \mathsf{wrp}_A.(y := g\ ;\ x := f) \\
\leq & \quad \{\text{Semantics of composition, take} \\
 & \quad\ \mathsf{wrp}_A.x := e \text{ for } P \text{ resp. } \mathsf{wrp}_A.x := f \text{ for } Q \\
 & \quad\ \text{and distribute the conditional with Lemma 8.1.4}\} \\
 & \mathsf{wrp}_A.y := g\ ;\ (\mathsf{wrp}_A.x := e \lhd b/A \rhd \mathsf{wrp}_A.x := f) \\
= & \quad \{\text{Semantics of conditional and composition}\} \\
 & \mathsf{wrp}_A.(y := g\ ;\ \mathsf{if}\ b\ \mathsf{then}\ x := e\ \mathsf{else}\ x := f\ \mathsf{fi})\ ,
\end{array}
$$

which finally proves

Theorem 8.1.2 (Correctness of code motion). If $x \neq y$, g is independent of x and b, e, f are independent of y, and if $\mathsf{in}_A(b) \cup \mathsf{in}_A(e) \cup \mathsf{in}_A(f) \subseteq \mathsf{in}_A(g) \cup \mathsf{def}(g)$, then

$$\begin{aligned}&\mathsf{wrp}_A.(\mathsf{if}\ b\ \mathsf{then}\ x := e\ ;\ y := g\ \mathsf{else}\ x := f\ ;\ y := g\ \mathsf{fi})\\ &\quad \leq \mathsf{wrp}_A.(y := g\ ;\ \mathsf{if}\ b\ \mathsf{then}\ x := e\ \mathsf{else}\ x := f\ \mathsf{fi})\ .\end{aligned}$$

□

To return to the comments made in Sect. 2.2 we mention that code motion (in the above example!) is correct in the sense of PTC because $\mathsf{in}_\emptyset(b) = \mathsf{in}_\emptyset(e) = \mathsf{in}_\emptyset(f) = \mathtt{false}$. Code motion is also correct in the PPC case as $\mathsf{in}_\Omega(g) \cup \mathsf{def}(g) = \mathtt{true}$ but note that the actual outcomes produced may differ in both programs. As mentioned in Sect. 2.2 it might make a difference if this is desirable. Furthermore the programs considered here are simplified versions; the situation obviously becomes harder if the branches do not solely consist of assignments, cf. Fig. 2.2.

8.1.3 Unswitching

More interesting is the unswitching strategy because piece of code is moved out of a loop. We consider the following two little programs:

$$\pi_1 \ = \ \mathsf{while}\ b\ \mathsf{do}\ \mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi}\ \mathsf{od}$$

and

$$\pi_2 \ = \ \mathsf{if}\ c\ \mathsf{then}\ \mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}\ \mathsf{else}\ \mathsf{while}\ b\ \mathsf{do}\ \pi'\ \mathsf{od}\ \mathsf{fi}\ ,$$

where π and π' are arbitrary programs. If neither π nor π' modify anything on which the guard c depends the translation of π_1 to π_2 seems reasonable and quite more efficient because the conditional has to be executed just once in the beginning and not each time the loop is iterated. This subsection is devoted to a precise analysis.

First of all, just like before, we have to formalize what it means that a program leaves a Boolean expression untouched because this is the essential requirement in order to let unswitching become sound. The adequate adjective is the following (see, e.g., [8]): A program π is said to be *transparent* to a Boolean expression c iff

$$\mathsf{in}_O(c) \ \subseteq \ \mathsf{wlp}.\pi.(\mathsf{in}_O(c))\ , \tag{8.2}$$

i.e. $\{\mathsf{in}_O(c)\}\pi\{\mathsf{in}_O(c)\}$, for all $O \subseteq \Sigma$. The notion of transparency suggests that the Boolean expression c does not notice the presence of program π as the evaluation of c does not depend on outcomes produced by π. More formally, the above definition can be justified by the little calculation:

$$\begin{aligned}&\mathsf{in}_O(c) \ \subseteq \ \mathsf{wlp}.\pi.(\mathsf{in}_O(c))\\ \Longleftrightarrow\quad &\{\text{Definitions}\}\\ &\forall s :\ \mathcal{B}(c)(s) \in O :\ (\forall \sigma :\ (s,\sigma) \in R(\pi) :\ \sigma \in \Omega \cup \mathsf{in}_O(c))\\ \Longleftrightarrow\quad &\{\text{Nesting, definition of } \mathsf{in}_O(c), \text{ naming conventions}\}\\ &\forall s, s' :\ \mathcal{B}(c)(s) \in O \wedge (s,s') \in R(\pi) :\ \mathcal{B}(c)(s') \in O\ .\end{aligned}$$

Operationally speaking, if c evaluates to a certain outcome (note that (8.2) has to hold for all $O \subseteq \Sigma \cup \Omega$, hence also for singletons) before execution of π then c

evaluates to the same outcome after execution of π, of course only if π delivers a regular result so that c can be evaluated at all.

Before the actual discussion some abbreviations are to be introduced in order to increase readability. For the present purposes it is convenient to denote the conditional resp. loop, $\mathsf{if}\ b\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi}$ resp. $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$, by the less clumsy $\pi \subset b \supset \pi'$ resp. $b * \pi$. Furthermore, some more observations are of independent interest so we start to collect yet some more rules.

It is an easy exercise – and we thus leave it to the reader – to see that an assignment, say, $x := e$ is transparent to c if c is independent of x. This suggests to look for similar but more general distribution properties. So assume P and Q are weakest relative precondition predicate transformers. Then, for all ψ,

$$
\begin{array}{ll}
 & (\mathsf{wrp}_A.\pi\ ;\ (P \lhd c/A \rhd Q)).\psi\ \cap\ (c = \mathsf{tt}) \\
= & \qquad \{\text{Application and definition of semantical conditional}\} \\
 & \mathsf{wrp}_A.\pi.(\mathsf{in}_A(c) \cup ((c = \mathsf{tt}) \cap P.\psi) \cup ((c = \mathsf{ff}) \cap Q.\psi))\ \cap\ (c = \mathsf{tt}) \\
\subseteq & \qquad \{\text{Assume } \pi \text{ is transparent to } c\} \\
 & \mathsf{wrp}_A.\pi.(\mathsf{in}_A(c) \cup ((c = \mathsf{tt}) \cap P.\psi) \cup ((c = \mathsf{ff}) \cap Q.\psi))\ \cap\ \mathsf{wlp}.\pi.(c = \mathsf{tt}) \\
= & \qquad \{\text{Generalized pairing condition and distribute}\} \\
 & \mathsf{wrp}_A.\pi.((c = \mathsf{tt}) \cap P.\psi) \\
\subseteq & \qquad \{\text{Monotonicity and definition of ';'}\} \\
 & (\mathsf{wrp}_A.\pi\ ;\ P).\psi\ .
\end{array}
$$

If π' is also assumed to be transparent to c this little calculation is the key to the following observation.

Lemma 8.1.5 (Distribute conditional and transparency). If π and π' are transparent to c, then

$$
\begin{array}{l}
(\mathsf{wrp}_A.\pi\ ;\ (P \lhd c/A \rhd Q)) \lhd c/A \rhd (\mathsf{wrp}_A.\pi'\ ;\ (P \lhd c/A \rhd Q)) \\
\quad \leq (\mathsf{wrp}_A.\pi\ ;\ P) \lhd c/A \rhd (\mathsf{wrp}_A.\pi'\ ;\ Q)\ .
\end{array}
$$

□

Now, consider the following situation which will be the next exercise. Suppose P, Q and R are weakest relative precondition predicate transformers; is it the case that

$$
\begin{array}{l}
(P \lhd c/A \rhd Q) \lhd b/A \rhd R \\
\quad \leq (P \lhd b/A \rhd R) \lhd c/A \rhd (Q \lhd b/A \rhd R)\ ?
\end{array}
$$

In some sense this distribution looks auspicious – just check the possible paths like in a binary tree – but it is enormously important to respect the presence of finite errors. Let us start to unroll the left hand side

$$
\begin{array}{ll}
 & ((P \lhd c/A \rhd Q) \lhd b/A \rhd R).\psi \\
= & \qquad \{\text{Definition of semantical conditional twice and distribute}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{in}_A(b) \cup \\
((b = \mathsf{tt}) \cap \mathsf{in}_A(c)) \cup \\
((b = \mathsf{tt}) \cap (c = \mathsf{tt}) \cap P.\psi) \cup \\
((b = \mathsf{tt}) \cap (c = \mathsf{ff}) \cap Q.\psi) \cup \\
((b = \mathsf{ff}) \cap R.\psi) \ ,
\end{array}
$$

and similarly the right hand side

$$
\begin{array}{ll}
 & ((P \lhd b/A \rhd R) \lhd c/A \rhd (Q \lhd b/A \rhd R)).\psi \\
= & \quad \{\text{Definition of semantical conditional thrice and distribute}\} \\
 & \mathsf{in}_A(c) \cup \\
 & ((c = \mathsf{tt}) \cap \mathsf{in}_A(b)) \cup \\
 & ((c = \mathsf{tt}) \cap (b = \mathsf{tt}) \cap P.\psi) \cup \\
 & ((c = \mathsf{tt}) \cap (b = \mathsf{ff}) \cap R.\psi) \cup \\
 & ((c = \mathsf{ff}) \cap \mathsf{in}_A(b)) \cup \\
 & ((c = \mathsf{ff}) \cap (b = \mathsf{tt}) \cap Q.\psi) \cup \\
 & ((c = \mathsf{ff}) \cap (b = \mathsf{ff}) \cap R.\psi) \ .
\end{array}
$$

To ensure that the upper predicate "implies" the lower some further assumptions concerning irregular outcomes are needed, namely

$$\mathsf{in}_A(b) \cup (b = \mathsf{ff}) \ \subseteq \ \mathsf{def}(c) \cup \mathsf{in}_A(c) \ .$$

This requirement is natural because it reflects that in the former program the guard b is evaluated first: If b evaluates to false or to an accepted outcome than evaluation of c must either yield an accepted outcome, then everything is fine, or must be defined such that evaluation of b follows immediately in the latter program. We summarize this observation in

Lemma 8.1.6 (Shuffle conditionals). If $\mathsf{in}_A(b) \cup (b = \mathsf{ff}) \ \subseteq \ \mathsf{def}(c) \cup \mathsf{in}_A(c)$, then

$$(P \lhd c/A \rhd Q) \lhd b/A \rhd R \ \leq \ (P \lhd b/A \rhd R) \lhd c/A \rhd (Q \lhd b/A \rhd R) \ .$$

□

After these preparations let us come to the actual "unswitching". As loops are present, showing $\mathsf{wrp}_A.\pi_1 \leq \mathsf{wrp}_A.\pi_2$ depends on whether divergence is accepted or not. We start with a variation of preservation of total correctness so assume $\infty \notin A$. By definition of the semantics of a loop, least fixpoints are to be taken, and in our scenario the claim reads as follows:

$$\mu\mathcal{W}_{b,\mathsf{wrp}_A.(\mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi})} \ \leq \ \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu W_{b,\mathsf{wrp}_A.\pi'} \ . \tag{8.3}$$

The isolated fixpoint on the left hand side is screaming for an application of the induction rule. Therefore we calculate

$$
\begin{array}{ll}
& \mathcal{W}_{b,\mathsf{wrp}_A.(\mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi})}(\mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'}) \\
= & \quad \{\text{Definition of } \mathcal{W}\} \\
& (\mathsf{wrp}_A.(\mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi})\ ;\ (\mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'})) \\
& \lhd b/A \rhd Id \\
= & \quad \{\text{Semantics of conditional}\} \\
& ((\mathsf{wrp}_A.\pi \lhd c/A \rhd \mathsf{wrp}_A.\pi')\ ;\ (\mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'})) \\
& \lhd b/A \rhd Id \\
= & \quad \{\text{Distribute conditional, general case (Lemma 7.2.1)}\} \\
& ((\mathsf{wrp}_A.\pi\ ;\ (\mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'})) \\
& \quad \lhd c/A \rhd \\
& \quad (\mathsf{wrp}_A.\pi'\ ;\ (\mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'}))) \\
& \lhd b/A \rhd Id \\
\leq & \quad \{\text{Distribute conditional again, here assume} \\
& \qquad \text{that } \pi \text{ and } \pi' \text{ are transparent to } c \text{ (Lemma 8.1.5)}\} \\
& ((\mathsf{wrp}_A.\pi\ ;\ \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi}) \lhd c/A \rhd (\mathsf{wrp}_A.\pi'\ ;\ \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'})) \lhd b/A \rhd Id \\
\leq & \quad \{\text{Shuffle conditionals under appropriate assumptions}\} \\
& ((\mathsf{wrp}_A.\pi\ ;\ \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi}) \lhd b/A \rhd Id) \\
& \lhd c/A \rhd \\
& ((\mathsf{wrp}_A.\pi'\ ;\ \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'}) \lhd b/A \rhd Id) \\
= & \quad \{\text{Definition of } \mathcal{W} \text{ and roll the fixpoint}\} \\
& \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi} \lhd c/A \rhd \mu\mathcal{W}_{b,\mathsf{wrp}_A.\pi'}\ .
\end{array}
$$

Thus, under the assumption that π and π' are transparent to c and that $\mathsf{in}_A(b) \cup (b = \mathsf{ff}) \subseteq \mathsf{def}(c) \cup \mathsf{in}_A(c)$ an application of the induction rule indeed implies the current goal (8.3) where $\infty \notin A$.

What remains is the variant of preservation of partial correctness where divergence is accepted. Here, loops are defined via greatest fixpoints and the same recipe as above does not work as the induction rule does not apply in the first step. However, the claim can be proved as follows. We start with massaging the goal,

$$
\begin{array}{ll}
& \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\
& \quad \subseteq \mathsf{wrp}_A.(b * \pi) \subset c \supset (b * \pi').\psi \\
\iff & \quad \{\text{Semantics of conditional}\} \\
& \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\
& \quad \subseteq \mathsf{in}_A(c) \cup \\
& \qquad ((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * \pi.\psi) \cup \\
& \qquad ((c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * \pi'.\psi) \\
\iff & \quad \{\text{Shunt, } \neg\mathsf{in}_A(c) = \mathsf{in}_{\Omega\setminus A}(c) \cup (c = \mathsf{tt}) \cup (c = \mathsf{ff}), \text{ distribute}\}
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathsf{in}_{\Omega\setminus A}(c) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \cup \\
 & ((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \cup \\
 & ((c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \\
 & \quad \subseteq ((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * \pi.\psi) \cup \\
 & \quad\quad ((c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * \pi'.\psi) \\
\Longleftarrow & \quad \{\text{Assume for the moment that} \\
 & \quad\quad \mathsf{in}_{\Omega\setminus A}(c) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi = \mathtt{false}\} \\
 & (c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \subseteq (c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * \pi.\psi \\
 & \text{and} \\
 & (c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \subseteq (c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * \pi'.\psi \\
\Longleftrightarrow & \quad \{\text{Obvious}\} \\
 & (c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \subseteq \mathsf{wrp}_A.b * \pi.\psi \\
 & \text{and} \\
 & (c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \subseteq \mathsf{wrp}_A.b * \pi'.\psi \quad ,
\end{array}
$$

and get a stronger formulation on the predicate level. Now the predicate transformers for loops are on the "right" side to apply the induction rule in principle, but notice that, unfortunately, these predicate transformers are applied to an argument so simple fixpoint rules still do not apply. However, the following observation is the key to a remedy.

It turns out that, for a predicate ψ, the function $(.\psi) \in (\mathit{PTrans} \to \mathit{Pred})$ which maps an $f \in \mathit{PTrans}$ to $f.\psi$ is universally conjunctive. To see this we look for $(.\psi)$'s lower adjoint, see Sect. 3.2, and define function $g_\psi \in (\mathit{Pred} \to \mathit{PTrans})$ pointwisely as follows: For $\phi, \chi \in \mathit{Pred}$:

$$s \in (g_\psi.\phi).\chi \overset{\text{def}}{\Longleftrightarrow} s \in \phi \wedge \psi \subseteq \chi \quad .$$

Then it is easy to show

$$\phi \subseteq (.\psi).f \quad \Longleftrightarrow \quad g_\psi.\phi \leq f$$

for all $\phi \in \mathit{Pred}$ and $f \in \mathit{PTrans}$.
In "$\Longrightarrow$" one assumes $\phi \subseteq (.\psi).f$, i.e. $\phi \subseteq f.\psi$, and takes arbitrary $\chi \in \mathit{Pred}$ and $s \in \Sigma$ with $s \in (g_\psi.\phi).\chi$. It follows that $s \in \phi$ and $\psi \subseteq \chi$ such that $s \in f.\psi$ by assumption and finally $s \in f.\chi$ by monotonicity. Thus, $g_\psi.\phi \leq f$ as required.
For "$\Longleftarrow$" one takes $g_\psi.\phi \leq f$ for granted and takes an arbitrary $s \in \phi$. As obviously $\psi \subseteq \psi$ it follows that $s \in (g_\psi.\phi).\psi$ by definition and thus $s \in f.\psi$ by assumption. In other words, $\phi \subseteq (.\psi).f$, and this completes the proof.

As the transfer lemma, cf. (3.7), suggests greatest fixpoint harmonize with universally conjunctive functions well. This is particularly the case here: For an arbitrary $f \in \mathit{PTrans}$ it is obvious that

$$
\begin{array}{l}
((\mathsf{wrp}_A.\pi \;;\; f) \lhd b/A \rhd \mathit{Id}).\psi \\
\quad = \mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.(f.\psi)) \cup ((b = \mathsf{ff}) \cap \psi) \quad ,
\end{array}
$$

or equivalently,

$$(.\psi)((\mathsf{wrp}_A.\pi\ ;\ f) \lhd b/A \rhd Id)$$
$$= \mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.((.\psi).f) \cup ((b = \mathsf{ff}) \cap \psi)\ \ ,$$

Thus, the transfer lemma ensures that

$$(.\psi)(\nu_X((\mathsf{wrp}_A.\pi\ ;\ X) \lhd b/A \rhd Id))$$
$$= \nu_\phi(\ \mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.\phi) \cup ((b = \mathsf{ff}) \cap \psi)\ )\ \ .$$

Though not required here, we like to mention that a very similar calculation establishes a corresponding result for least fixpoints: Function $(.\psi)$ is also universally disjunctive – $g_\psi \in (Pred \to PTrans)$ defined by $s \in (g_\psi.\phi).\chi \stackrel{\text{def}}{\iff} \chi \subseteq \psi \to s \in \phi$ is its upper adjoint – and the transfer lemma for least fixpoints (3.6) applies analogously. This seems to be a noteworthy result as it confirms the intuition on the predicate level.

Theorem 8.1.3 (Loop's semantics on the predicate level).

$$\mathsf{wrp}_A.(\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}).\psi$$
$$= \lambda\phi(\ \mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.\phi) \cup ((b = \mathsf{ff}) \cap \psi)\ )\ \ ,$$

where $\lambda = \nu$ if $\infty \in A$ and $\lambda = \mu$ otherwise.
□

Now it is clear that the loops are indeed on the "right" side to let simpler fixpoint rules apply – just lower the fixpoints to the predicate level before – and we can continue the calculation as follows. But first of all we have to justify the last but one step in the prior calculation where the very upper disjunct vanishes. We start with the second conjunct thereof

$$\begin{array}{ll} & \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\ = & \quad \{\text{Lower the fixpoint and unroll it, definitions}\} \\ & \mathsf{in}_A(b) \cup \\ & ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi \subset c \supset \pi'.(\mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi)) \cup \\ & ((b = \mathsf{ff}) \cap \psi) \\ \subseteq & \quad \{\text{Semantics of conditional, monotonicity}\} \\ & \mathsf{in}_A(b) \cup \\ & ((b = \mathsf{tt}) \cap (\mathsf{in}_A(c) \cup (c = \mathsf{tt}) \cup (c = \mathsf{ff}))) \cup \\ & (b = \mathsf{ff}) \end{array}$$

such that finally, together with the first conjunct,

$$\mathsf{in}_{\Omega\setminus A}(c) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi\ \ \subseteq\ \ \mathsf{in}_{\Omega\setminus A}(c) \cap (\mathsf{in}_A(b) \cup (b = \mathsf{ff}))$$

remains. The right hand side equals `false`, thus justifies the last but one step, precisely if

$$\mathsf{in}_A(b) \cup (b = \mathsf{ff})\ \ \subseteq\ \ \mathsf{in}_A(c) \cup \mathsf{def}(c)\ \ ,$$

and luckily this requirement is not new, we already needed it in the previous case. Hence, let us assume the same here.

Furthermore, we are left with showing, e.g.,

$$(c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \ \subseteq \ \mathsf{wrp}_A.b * \pi.\psi$$

or equivalently, remember that divergence is accepted such that loops are defined via greatest fixpoints,

$$\begin{array}{l}(c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\ \quad \subseteq \nu_X(\mathsf{wrp}_A.\pi \ ; X \lhd b/A \rhd Id).\psi \ ,\end{array}$$

which is equivalent to

$$\begin{array}{l}(c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\ \quad \subseteq \nu\phi(\mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.\phi) \cup ((b = \mathsf{ff}) \cap \psi))\end{array}$$

by Theorem 8.1.3. From here we continue as shown below.

$$\begin{array}{ll}
 & (c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\
 & \quad \subseteq \nu\phi(\mathsf{in}_A(b) \cup ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.\phi) \cup ((b = \mathsf{ff}) \cap \psi)) \\
\Longleftarrow & \quad \{\text{Induction rule}\} \\
 & (c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \\
 & \quad \subseteq \mathsf{in}_A(b) \cup \\
 & \quad\quad ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi)) \cup \\
 & \quad\quad ((b = \mathsf{ff}) \cap \psi) \\
\Longleftrightarrow & \quad \{\text{Lower left fixpoint (Theorem 8.1.3),} \\
 & \quad\quad \text{unroll it and distribute}\} \\
 & ((c = \mathsf{tt}) \cap \mathsf{in}_A(b)) \cup \\
 & ((c = \mathsf{tt}) \cap (b = \mathsf{tt}) \cap \\
 & \ \mathsf{wrp}_A.(\pi \subset c \supset \pi').(\mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi)) \cup \\
 & ((c = \mathsf{tt}) \cap (b = \mathsf{ff}) \cap \psi) \\
 & \quad \subseteq \mathsf{in}_A(b) \cup \\
 & \quad\quad ((b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi)) \cup \\
 & \quad\quad ((b = \mathsf{ff}) \cap \psi) \ .
\end{array}$$

We let 'rhs' denote the right-hand-side of the latter inclusion. From here, three sub-cases remain to be shown.

a) For the first disjunct one obviously has

$$(c = \mathsf{tt}) \cap \mathsf{in}_A(b) \subseteq \mathsf{in}_A(b) \subseteq \text{'rhs'} \ ,$$

and similarly for the third conjunct, i.e.

b), where

$$(c = \mathsf{tt}) \cap (b = \mathsf{ff}) \cap \psi \subseteq (b = \mathsf{ff}) \cap \psi \subseteq \text{'rhs'} \ .$$

c) The final, the second, disjunct is the most interesting, and here one calculates

$$
\begin{array}{ll}
 & (c = \mathsf{tt}) \cap (b = \mathsf{tt}) \cap \mathsf{wrp}_A.(\pi \subset c \supset \pi').(\mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \\
\subseteq & \quad \{\text{Semantics of conditional}\} \\
 & (c = \mathsf{tt}) \cap (b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.(\mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \\
\subseteq & \quad \{\text{As before, assume } \pi \text{ is transparent to } c\} \\
 & \mathsf{wlp}.\pi.(c = \mathsf{tt}) \cap (b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.(\mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \\
\subseteq & \quad \{\text{Generalized pairing condition, i.e. Lemma 6.1.1}\} \\
 & (b = \mathsf{tt}) \cap \mathsf{wrp}_A.\pi.((c = \mathsf{tt}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi) \\
\subseteq & \quad \{\text{Obvious}\} \\
 & \text{`rhs'} \ ,
\end{array}
$$

which completes this case. To show the missing part, i.e.

$$(c = \mathsf{ff}) \cap \mathsf{wrp}_A.b * (\pi \subset c \supset \pi').\psi \subseteq \mathsf{wrp}_A.b * \pi'.\psi \ ,$$

one argues completely analogously, just use the fact that π' is transparent to c, too.

Altogether, both calculations prove the overall claim of this subsection which reads, after resolving our condensed notations, as shown below.

Theorem 8.1.4 (Correctness of unswitching). If $\mathsf{in}_A(b) \cup (b = \mathsf{ff}) \subseteq \mathsf{in}_A(c) \cup \mathsf{def}(c)$ and if π and π' are transparent to c, then

$$
\begin{array}{l}
\mathsf{wrp}_A.\mathsf{while}\ b\ \mathsf{do}\ \mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi}\ \mathsf{od} \\
\quad \leq \mathsf{wrp}_A.\mathsf{if}\ c\ \mathsf{then}\ \mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}\ \mathsf{else}\ \mathsf{while}\ b\ \mathsf{do}\ \pi'\ \mathsf{od}\ \mathsf{fi}\ .
\end{array}
$$

□

Again, we like to make the concluding remarks that unswitching is also correct in the PPC-case as $\mathsf{in}_\Omega(c) \cup \mathsf{def}(c) = \mathtt{true}$. For PTC one needs $(b = \mathsf{ff}) \subseteq \mathsf{def}(c)$ but this does not universally hold. Consider, for instance, the following example. Suppose given the predicate $\{s\}$, i.e. the state s, with $\mathcal{B}(b)(s) = \mathsf{ff}$ but $\mathcal{B}(c)(s) \in \Omega$. Then

$$[\{s\}]\mathsf{while}\ b\ \mathsf{do}\ \mathsf{if}\ c\ \mathsf{then}\ \pi\ \mathsf{else}\ \pi'\ \mathsf{fi}\ \mathsf{od}[\mathtt{true}] \ ,$$

i.e. the loop terminates immediately, but it is *not* the case that

$$[\{s\}]\mathsf{if}\ c\ \mathsf{then}\ \mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}\ \mathsf{else}\ \mathsf{while}\ b\ \mathsf{do}\ \pi'\ \mathsf{od}\ \mathsf{fi}[\mathtt{true}]$$

because the conditional terminates immediately, but irregularly. Hence, unswitching does *not* preserve total correctness! Note that this situation does not occur in simpler scenarios where the presence of finite errors is ignored resp. where expression evaluations are assumed to terminate regularly. In this sense, wrp has truly kept its promise to facilitate reasoning about "phenomena" like these resp. to bring to light them at all.

8.2 Translation Verification

The task this section is concerned with reads rather simple: Given a program π of the high-level language, Sect. 7.4, translate it to a program π' of the assembly

language, Sect. 7.3, in such ways that the latter implements the former in the sense of preservation of relative correctness w.r.t. some set A of accepted outcomes. As the target language, i.e. the assembly language, is assumed to run on a finite machine which expresses in spontaneous occurrences of "StackOverflow"-errors we decide to accept this particular failure because we keep it for unavoidable. Moreover we like to exploit the properties that distinguish wrp from wp or wlp: The translation to be discussed soon will translate non-diverging source programs to non-diverging target programs but wlp cannot express this; on the other hand wp identifies divergence and runtime-errors and, therefore, it cannot treat this scenario in the presence of finite errors either. In fact, the "StackOverflow"-error is the only one which is due to some resource limitations and which is modeled in the semantics but it is straightforward to integrate others. There are other finite errors in the semantics, "ProcUndecl" and "EmptyStack", but they were mainly integrated because they are obvious candidates and in order to make the semantics slightly more exciting. However, it turns out that the translation scheme given below guarantees that non-closed programs will not be translated at all – for the "interesting cases", see p. 109 – resp. that translated programs will never execute a ret-command on an empty stack.

A more interesting question is whether divergence is to be tolerated or not, i.e. whether $\infty \in A$ or not. Both cases are of interest for particular applications. If one accepts divergence the proof obligations for the source program are a much easier because a termination proof for the source program is dispensable. This view is interesting in particular when proving compilers correct because it is hardly possible to guarantee that a compiler terminates for all its inputs and, moreover, it generally suffices that a compiler delivers a result just once. However, a compiler that constantly maps each given source program to a diverging target program preserves partial correctness by definition but this compiler is rather senseless (but we like to say that compiler construction is an engineering task and an honest programmer will try to avoid this). Hence, divergence might be kept for intolerable because one really wants the target program to terminate whenever the source program does (apart from finite errors which are due to some resource limitations, we already discussed this). This view is the one which is of interest for process programming purposes (outside safety critical systems); it allows to infer termination of the target program if termination is proved for the source program. Then, a termination proof for the latter has to be performed but that is worth in the sense just mentioned.

All in all, the claim of this section is that target program π' implements source program π in the sense of preservation of relative correctness w.r.t. $\{$"StackOverflow"$\}$ and also w.r.t. $\{\infty,$ "StackOverflow"$\}$. Two proofs have to be performed because, as observed before, preservation of total and preservation of partial correctness are independent notions.

We remark that a slightly modified version of the essence of the first claim, the one with $A = \{$"StackOverflow"$\}$, is already presented in [63]. There, due to lack of space, we focused on the most interesting cases, i.e. the procedure mechanism, and

this is the place for a more profound discussion and for a proof of the remaining cases.

8.2.1 Compiling specification

In Table 8.1 we inductively define a compiling relation $\mathcal{C} \subseteq \Pi \times MP \times TDict$.[4] Here, $TDict$ is the set of stacks $\boldsymbol{\delta}$ of *translation dictionaries*

$$\delta \in (ProcName \xrightarrow{\text{fin}} Lab) \ ,$$

each of which intuitively maps procedure names to labels where code for the corresponding body can be found. We adopt most of the notational conventions made for procedure dictionaries $PDict$, see p. 104. The empty translation dictionary is given by the empty set, $\emptyset$. Concatenation of translation dictionaries is denoted by an infix dot and we say that $\delta_i \in \boldsymbol{\delta}$ iff $\boldsymbol{\delta} = \delta_1 \cdot \ldots \cdot \delta_i \cdot \ldots \cdot \delta_n$. Again, $p \in \mathrm{dom}(\boldsymbol{\delta})$ means that there exists a dictionary $\delta \in \boldsymbol{\delta}$ with $p \in \mathrm{dom}(\delta)$. The stack of dictionaries consisting of all dictionaries up to index i is denoted by $\boldsymbol{\delta_i}$. Applying a stack of dictionaries $\boldsymbol{\delta} = \delta_1 \cdot \ldots \cdot \delta_n$ to an argument means to look up the topmost entry, i.e.

$$\boldsymbol{\delta_n}(p) \stackrel{\text{def}}{=} \begin{cases} \delta_n(p) & : \ p \in \mathrm{dom}(\delta_n) \\ \boldsymbol{\delta_{n-1}}(p) & : \ p \notin \mathrm{dom}(\delta_n) \wedge n > 1 \\ \text{undefined} & : \ \text{otherwise} \end{cases} \ , \tag{8.4}$$

this convention has to be made because, in contrast to procedure dictionaries, translation dictionaries will not be distinguished. As a consequence, the index j_p as defined on p. 105 is generally not unique. However, if $p \in \mathrm{dom}(\boldsymbol{\delta})$ then there exists a dictionary $\delta \in \boldsymbol{\delta}$ with $p \in \mathrm{dom}(\delta)$ by convention and from all those dictionaries having this property we define the topmost to have index j_p. With the aid of specification (8.4) above we can more formally say that $\boldsymbol{\delta}(p) = \delta_{j_p}(p)$ if $\boldsymbol{\delta}(p)$ is defined. Finally, for the sake of brevity we call a stack of translation dictionaries $\boldsymbol{\delta}$ also a dictionary for short.

Vocalizing $\mathcal{C}(\pi, m, \boldsymbol{\delta})$ means that assembly program m is a possible compiling result of source program π assuming that dictionary $\boldsymbol{\delta}$ assigns appropriate labels to the used procedure names. The program

$$\mathsf{seq}(\mathsf{assign}(\mathsf{y}, 1), \mathsf{while}(\mathsf{x} > 0, \mathsf{seq}(\mathsf{assign}(\mathsf{y}, \mathsf{x} * \mathsf{y}), \mathsf{assign}(\mathsf{x}, \mathsf{x} - 1)))) \ ,$$

for instance, which computes the factorial of x leaving the result in variable y, may irrespective of the used dictionary be compiled to the assembly program

$$\mathsf{asg}(\mathsf{y}, 1) \cdot \mathsf{Loop} \cdot \mathsf{cj}(\mathsf{x} \neq 0, \mathsf{End}) \cdot \mathsf{asg}(\mathsf{y}, \mathsf{x} * \mathsf{y}) \cdot \mathsf{asg}(\mathsf{x}, \mathsf{x} - 1) \cdot \mathsf{goto}(\mathsf{Loop}) \cdot \mathsf{End}$$

that we already know from Sect. 7.3.

Note that the typing constraint $m \in MP$ guarantees that target programs are labeled uniquely. Rule [Seq], for instance, can only be applied if the labels used in m_1 and m_2 are distinct. Obviously target programs must be labeled uniquely –

[4] We desist from translating the skip command as this is a rather obvious task; it is an ease to show that the results presented below transfer if it is translated to an unused label or to an assignment $\mathsf{asg}(x, x)$ which play the role of a "NoOp" in the real world.

[Assign] $\mathcal{C}(\mathsf{assign}(x,e), \mathsf{asg}(x,e), \boldsymbol{\delta})$

[Seq] $\dfrac{\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta}),\ \mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})}{\mathcal{C}(\mathsf{seq}(\pi_1,\pi_2), m_1 \cdot m_2, \boldsymbol{\delta})}$

[If] $\dfrac{\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta}),\ \mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})}{\mathcal{C}(\mathsf{if}(b,\pi_1,\pi_2), \mathsf{cj}(b,l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, \boldsymbol{\delta})}$

[While] $\dfrac{\mathcal{C}(\pi, m, \boldsymbol{\delta})}{\mathcal{C}(\mathsf{while}(b,\pi), l_0 \cdot \mathsf{cj}(b,l_1) \cdot m \cdot \mathsf{goto}(l_0) \cdot l_1, \boldsymbol{\delta})}$

[Call] $\dfrac{p \in \mathrm{dom}(\boldsymbol{\delta})}{\mathcal{C}(\mathsf{call}(p), \mathsf{jsr}(\boldsymbol{\delta}(p)), \boldsymbol{\delta})}$

[Blk] $\dfrac{\mathcal{C}(\pi_p, m_p, \boldsymbol{\delta} \cdot \{p \mapsto l_p\}),\ \mathcal{C}(\pi_b, m_b, \boldsymbol{\delta} \cdot \{p \mapsto l_p\})}{\mathcal{C}(\mathsf{blk}(p,\pi_p,\pi_b), \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, \boldsymbol{\delta})}$

Table 8.1. Compiling specification: Inductively relating source and target programs.

keep in mind that in the source language procedures may be redeclared blockwise – and defining a compiling relation enables us to solve this task elegantly. Note furthermore, that the rules do not otherwise restrict the choice of labels. It is an advantage of a relational specification over a compiling function, e.g. $C : \mathit{Prog} \times \mathit{TDict} \to \mathit{MP}$, that certain aspects, like choice of labels here, can be left open for a later design stage of the compiler. Defining a function would force us to prove unique labeling right now and this is a part beneath our considerations as we are interested in the implementation of the control flow only.

The compilation-rules should be intuitively clear but let us run over and comment on them as certain details are worth mentioning. The [Assign]-rule translates assignments to assignments, just as they are. Indeed, both assignments have the same semantics so this rule will not harm. We are interested in the linearization of the control-flow and assignments are integrated into the languages in order to be able to write – more or less – meaningful programs. Assignments, and any kind of commands which have a direct effect on the state and which execute in a linear fashion, are thought to be macros which can be replaced by (sequences of) more concrete commands of a real machine in a later stage. The sequential composition of programs corresponds to the concatenation of the translated parts, see the [Seq]-rule. As remarked before it only applies if distinct labels are used in each translation of the parts the composed program consists of. The [If]-rule instructs to jump over the translation of the first component to the translation of the second component if the guard evaluates to ff and to skip the conditional jump otherwise such that the first component followed by an unconditional jump to the end of the conditional is to be executed. The needed labels are introduced accordingly; note that those labels have to be fresh because otherwise the compound would not be a well-typed assembly program. Observe furthermore that the labels introduced here have nothing to do with the labels kept in the dictionary $\boldsymbol{\delta}$ in question, the latter keeps labels for procedures only and not for jumps in general. Similarly loops are translated, cf. the [While]-rule. Depending on the evaluation of the guard the translated body followed by an unconditional jump to the beginning or a jump out of the loop is to be executed. If a procedure named p is declared in the source program, i.e. if $p \in \mathrm{dom}(\boldsymbol{\delta})$, the [Call]-rule applies and a

call of procedure p, $\mathsf{call}(p)$, is translated to a procedure-jump to the corresponding label kept in the dictionary $\boldsymbol{\delta}$ in question, $\mathsf{jsr}(\boldsymbol{\delta}(p))$. It is essential to note that the premise of the compiling rule [Call] guarantees that non-closed programs cannot be compiled with an empty dictionary. Those procedure identifiers enter the dictionary if blocks are translated, see the [Blk]-rule. If a block with body π_b introducing p with procedure body π_p is translated using dictionary $\boldsymbol{\delta}$ one firstly has to translate the components, π_p and π_b, using a dictionary that has a new binding for p. This proceeding should be intuitively clear after the section concerned with the semantics of the source-language, Sect. 7.4, because this technique *implements* static-scoping. The actual code consists of a jump to the translation of the body of the block, an appropriate label for the introduced procedure followed by the translation of the procedure-body itself and a return-command. Note that the translation-scheme could also have put the translation of the procedure-body – together with label and return – to the end of the sequence. Note furthermore that the compiling specification can be straightforwardly extended to blocks in which systems of mutually recursive procedures are declared. As already mentioned in the introduction of the language this would burden the notations – and oncoming proofs – a lot without bringing any further enlightenments.

This seems to be the appropriate place for the following remarks. The reader might wonder why stacks of translation dictionaries need not to be distinguished resp. why the operational semantics of the source language deals with fresh identifiers and renamed programs. The reason is rather obvious: The "generated" target programs are labeled uniquely. This is different for the operational semantics of the source language and, in some sense, renaming with fresh identifiers makes sure that the program is distinguished – what the procedure identifiers concerns – at execution time. On the other hand, an operational semantics for the source language which follows the shape of the compiling specification given here, in particular a modified definition of j_p via a lookup from the top, does not work for the following reasons. To execute a call a prefix stack must be used – in order to model static scoping – but then the blk and seq rules do not apply anymore because they require the entry- and exit-dictionaries to be equal. This is needed because the subsequent commands must use the entire stack of dictionaries in order to be able to call procedures which are defined later. A way out of this dilemma would be to make use of closures in such ways that procedures are completely executed before the subsequent instructions are started. But this was not what we intended to specify since we aimed at a purely structural operational semantics. However, we furthermore like to note that consequently the oncoming translation correctness proof does not resemble the equivalence proof given in Sect. 7.4. This comment is worth mentioning because parts of the equivalence proof can be kept for a correctness proof of the identity-translation from the source language to itself where given programs are supposed to have a denotational and (the same) generated programs are supposed to have an operational semantics.

8.2.2 Compiling correctness

This section is concerned with proving correctness of the translation specified just before. As already discussed, the translation cannot be correct in the sense of preservation of total correctness because our assembly language might report "StackOverflow" on executing a jsr instruction, and thus regularly terminating source programs might be compiled to target programs that do not terminate regularly. Nevertheless – as we will see in a while – source programs that do not diverge are never compiled to diverging target programs. But preservation of total correctness identifies divergence and runtime-errors and, therefore, it cannot treat this scenario appropriately. A main purpose of the story told here is to show how the greater selectivity of wrp-based reasoning allows a more adequate treatment of this scenario by an appropriate choice of A. Proving correct the translation in the sense of a variant of preservation of total correctness is, thus, an interesting case. The counterpart, the corresponding variant of preservation of partial correctness is, as motivated before, also worth while because it is easier to apply in practise since there are weaker proof obligations for the source program. Actually, the latter variant is the seemingly harder and less common part.

As a matter of fact, two theorems and consequently two proofs have to be given resp. performed because neither does a translation preserve partial correctness if it preserves total correctness nor vice versa. To be more specific we treat "StackOverflow" as an acceptable error because it may spontaneously occur every now and then and we simply cannot do anything against it. In a first case we will not accept divergence which gives rise to a relativized version of preservation of total correctness where $A = \{\text{"StackOverflow"}\}$ (this is the full version of our [63] if you like). In the second case we will accept divergence such that, with $A = \{\infty, \text{"StackOverflow"}\}$, a relativized version of preservation of partial correctness will be be discussed.

A variant of preservation of total correctness. We are going to prove the following theorem.

Theorem 8.2.1. Suppose $\infty \notin A$ and "StackOverflow" $\in A$. Then for all π and m:

$$\mathcal{C}(\pi, m, \emptyset) \;\Rightarrow\; \mathsf{wrp}_A.m \geq \mathsf{wrp}_A^{\perp_{Env}}.\pi \quad .$$

Here, $\perp_{Env}$ denotes the "undefined" environment that maps each procedure identifier to $\perp$, i.e. $\forall p \in \mathit{ProcName} :: \perp_{Env}(p) = \perp$. As mentioned in Sect. 7.3 this is reasonable because otherwise undefined procedures would miraculously have a non-trivial meaning. Thus, if a program π is compiled to an assembly program m in an empty dictionary, relative correctness is preserved in the sense that, very roughly speaking, the target program terminates regularly whenever the source program does and the results coincide, unless the machine is just too small to store the needed return-addresses. To be more accurate, if the source program is proved to be totally correct w.r.t. a pre- and a postcondition (note that "StackOverflow" is not present in the semantics of the source language) then, for every initial state

satisfying the precondition, the target program will terminate regularly in a state satisfying the postcondition, in particular this is a result which is also possible for the source program, but it may also terminate irregularly, i.e. abort, propagating a "StackOverflow". This property is of interest for process programming (outside safety-critical systems) as it allows to conclude the behavior of the target program from the behavior of the source program.

When we try to prove Theorem 8.2.1 by a structural induction – which is the most proximate approach – we are faced with two problems. Firstly, when machine programs are put together to implement composed programs, like in the [Seq] or [If] rule, the induction hypothesis cannot be directly applied because it is concerned with code for the components in isolation while, in the composed code, the code runs in the context of other code. In other words, Theorem 8.2.1 turns out to be improvable in the direct way because of its lack of compositionality. Our approach to deal with this problem is to establish a stronger claim that involves a universal quantification over all contexts.[5] More specifically, we show

$$\mathsf{wrp}_A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; \mathsf{wrp}_A.(u \cdot m, v, a)$$

for all surrounding code sequences u, v and stack contexts a. Note how the sequential composition with $\mathsf{wrp}_A(u \cdot m, v, a)$ on the right hand side beautifully expresses that m transfers control to the subsequent code and that the stack is left unchanged. Furthermore, Theorem 8.2.1 follows immediately with suitable instantiation.

Secondly, when considering the call-case, some knowledge about the bindings in the dictionary $\boldsymbol{\delta}$ is needed. We already know this problem from Sect. 7.4 where we had to formalize some expectations concerning the relationship between syntactic and semantic bindings in the semantics of the source language. Motivated by the shape of the bindings-predicates used there we try to solve the current problem with the aid of the following predicate which expresses appropriate expectations for the present task.

$$\begin{array}{l} \mathsf{fit}(\eta, \boldsymbol{\delta}, u) \quad \stackrel{\text{def}}{\Longleftrightarrow} \\ \quad \forall q \in \mathrm{dom}(\boldsymbol{\delta}) :: \;\; \exists x, y :: \\ \qquad x \cdot \boldsymbol{\delta}(q) \cdot y = u \;\; \wedge \\ \qquad \forall e, f, g :: \;\; \mathsf{wrp}_A.(x, \boldsymbol{\delta}(q) \cdot y, g \cdot \langle e, f \rangle) \geq \eta(q) \;;\; \mathsf{wrp}_A.(e, f, g) \;\; . \end{array} \tag{8.5}$$

It expresses that the bindings in translation dictionary $\boldsymbol{\delta}$ together with the assembly code u (that comprises the context in which the implementing code is running) "fit" to the bindings in the semantic environment η used by the source program. The first conjunct says that the context provides a corresponding label for each procedure q bound by $\boldsymbol{\delta}$; the second conjunct tells us that the code following this label implements q's binding in η and proceeds with the code on top of the return stack. This is just what is needed in the call-case of the induction. The code generated for blocks has to ensure that this property remains valid for newly declared procedures. Putting the pieces together we are thus going to prove

[5] This fundamental but very intuitive view on execution progress of assembly programs was presented in [60] at first. Sadly it has never been published for, say, unjustified reasons.

Lemma 8.2.1. Suppose $\infty \notin A$ and "StackOverflow" $\in A$. For all π ,m, u, v, a, η, $\boldsymbol{\delta}$:

$$\begin{aligned}&\mathcal{C}(\pi, m, \boldsymbol{\delta}) \wedge \mathsf{fit}(\eta, \boldsymbol{\delta}, u \cdot m \cdot v)\\ &\quad \Rightarrow \mathsf{wrp}_A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; \mathsf{wrp}_A.(u \cdot m, v, a) \;\;.\end{aligned}$$

Theorem 8.2.1 follows by the instantiation $u = v = \varepsilon$, $a = \varepsilon$, $\eta = \bot_{Env}$, $\boldsymbol{\delta} = \emptyset$ using the [Term-wrp] law and $\mathsf{wrp}_A.m = \mathsf{wrp}_A.(\varepsilon, m, \varepsilon)$.

Proof of Lemma 8.2.1. The proof is by structural induction on π. So consider some arbitrarily chosen $\pi, m, u, v, a, \eta, \boldsymbol{\delta}$ such that $\mathcal{C}(\pi, m, \boldsymbol{\delta})$ and $\mathsf{fit}(\eta, \boldsymbol{\delta}, u \cdot m \cdot v)$, and assume that for all component programs the claim of Lemma 8.2.1 holds. As usual, we proceed by a case analysis on the structure of π. In each case we perform a kind of "symbolic execution" of the corresponding assembly code using the wrp-laws from Sect. 7.3. The assumptions about fit will solve the call-case elegantly, the while- and blk-case moreover involve some fixpoint reasoning as the semantics of those commands is defined via fixpoints.

Case a.) $\pi = \mathsf{assign}(x, e)$. By the [Assign] compiling rule, $m = \mathsf{asg}(x, e)$ and

$$\begin{aligned}&\mathsf{wrp}_A.(u, \mathsf{asg}(x, e) \cdot v, a)\\ \geq\;& \quad \{\text{Law [Asg-wrp]}\}\\ &(x :=_A e) \;;\; \mathsf{wrp}_A.(u \cdot \mathsf{asg}(x, e), v, a)\\ =\;& \quad \{\text{Semantics of an assignment}\}\\ &\mathsf{wrp}_A^{\eta}.\mathsf{assign}(x, e) \;;\; \mathsf{wrp}_A.(u \cdot \mathsf{asg}(x, e), v, a) \;\;.\end{aligned}$$

Case b.) $\pi = \mathsf{seq}(\pi_1, \pi_2)$. By the [Seq] compiling rule, there are m_1, m_2 with $m = m_1 \cdot m_2$ such that $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$ and $\mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})$ holds. Then,

$$\begin{aligned}&\mathsf{wrp}_A.(u, m_1 \cdot m_2 \cdot v, a)\\ \geq\;& \quad \{\text{Induction hypothesis for } \pi_1\}\\ &\mathsf{wrp}_A^{\eta}.\pi_1 \;;\; \mathsf{wrp}_A.(u \cdot m_1, m_2 \cdot v, a)\\ \geq\;& \quad \{\text{Induction hypothesis for } \pi_2\}\\ &\mathsf{wrp}_A^{\eta}.\pi_1 \;;\; \mathsf{wrp}_A^{\eta}.\pi_2 \;;\; \mathsf{wrp}_A.(u \cdot m_1 \cdot m_2, v, a)\\ =\;& \quad \{\text{Semantics of composition}\}\\ &\mathsf{wrp}_A^{\eta}.\mathsf{seq}(\pi_1, \pi_2) \;;\; \mathsf{wrp}_A.(u \cdot m_1 \cdot m_2, v, a) \;\;.\end{aligned}$$

Case c.) $\pi = \mathsf{if}(b, \pi_1, \pi_2)$. By the [If] compiling rule, $m = \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2$ for certain m_1, m_2 with $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$ and $\mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})$. Here one calculates:

$$\begin{aligned}&\mathsf{wrp}_A.(u, \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a)\\ \geq\;& \quad \{\text{Law [Cj-wrp]}\}\\ &\mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1), m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a)\\ &\lhd b/A \rhd\\ &\mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2), l_1 \cdot m_2 \cdot l_2 \cdot v, a)\end{aligned}$$

$$
\begin{array}{ll}
\geq & \{\text{Induction hypothesis for } \pi_1 \text{ in the first,} \\
& \quad \text{law [Label-wrp] in the second component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1, \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1, m_2 \cdot l_2 \cdot v, a) \\
\geq & \{\text{Induction hypothesis for } \pi_2 \text{ in the second,} \\
& \quad \text{law [Goto-wrp] in the first component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2, l_2 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}^{\eta}_{A}.\pi_2 \ ; \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2, l_2 \cdot v, a) \\
\geq & \{\text{Law [Label-wrp] in both components}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}^{\eta}_{A}.\pi_2 \ ; \mathsf{wrp}_A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
= & \{\text{Distribute conditional, i.e. Lemma 7.2.1}\} \\
& (\mathsf{wrp}^{\eta}_{A}.\pi_1 \lhd b/A \rhd \mathsf{wrp}^{\eta}_{A}.\pi_2) \ ; \\
& \mathsf{wrp}_A(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
= & \{\text{Semantics of conditional}\} \\
& \mathsf{wrp}^{\eta}_{A}.\mathsf{if}(b, \pi_1, \pi_2) \ ; \mathsf{wrp}_A(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \ .
\end{array}
$$

Case d.) $\pi = \mathsf{while}(b, \pi_1)$. By the [While] compiling rule, $m = l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1$ for an m_1 with $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$. This is a more interesting case and the calculation starts with

$$
\begin{array}{ll}
& \mathsf{wrp}_A.(u, l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
\geq & \{\text{Laws [Label-wrp] and [Cj-wrp]}\} \\
& \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1), m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0), l_1 \cdot v, a) \\
\geq & \{\text{Induction hypothesis for } \pi_1 \text{ in the first,} \\
& \quad \text{law [Label-wrp] in the second component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1, \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a) \\
\geq & \{\text{Law [Goto-wrp] in the first component,}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}_A.(u, l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a) \ ,
\end{array}
$$

which reflects the intuition that, depending on the evaluation of the guard, either the body of the loop is executed once and one is standing right back at the beginning where the calculation starts, or one jumps out of the loop otherwise. Introducing an abbreviation for the "remainder",

$$R \;=\; \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a)$$

and defining the monotonic transformer $g : PTrans \to PTrans$ by

$$g(X) \;=\; (\mathsf{wrp}^{\eta}_A.\pi_1 \;;\; X) \lhd b/A \rhd R \;\;,$$

the above calculation has established

$$\mathsf{wrp}_A.(u, m \cdot v, a) \geq g(\mathsf{wrp}_A.(u, m \cdot v, a)) \;\;.$$

By the induction rule

$$(*) \quad \mathsf{wrp}_A(u, m \cdot v, a) \geq \mu g$$

follows immediately. Furthermore,

$$\begin{array}{ll} & g(X \;;\; R) \\ = & \qquad \{\text{Definition of } g\} \\ & (\mathsf{wrp}^{\eta}_A.\pi_1 \;;\; X \;;\; R) \lhd b/A \rhd R \\ = & \qquad \{\text{Distribute Conditional, } Id \text{ is the unit of composition}\} \\ & ((\mathsf{wrp}^{\eta}_A.\pi_1 \;;\; X) \lhd b/A \rhd Id) \;;\; R \\ = & \qquad \{\text{Definition of } \mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi_1}\} \\ & \mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi_1}(X) \;;\; R \;\;. \end{array}$$

Letting $(;\, R) : PTrans \to PTrans$ denote the transformer that sequentially composes R from the right, this calculation has shown

$$g((;\, R)(X)) \;=\; (;\, R)(\mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi_1}(X)) \;\;.$$

Since the transformer $(;\, R)$ is universally disjunctive, see (4.1), we conclude that

$$(**) \quad \mu g \;=\; (;\, R)(\mu \mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi_1})$$

by the transfer lemma, see (3.6). Resolving the definitions and abbreviations and combining $(*)$ with $(**)$ finally yields

$$\begin{array}{l} \mathsf{wrp}_A.(u, l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\ \quad \geq \mu \mathcal{W}_{b,\mathsf{wrp}^{\eta}_A.\pi_1} \;;\; \mathsf{wrp}_A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a) \end{array}$$

which is exactly the claim for this case, see the definition of the semantics of a loop.

Case e.) $\pi = \mathsf{call}(p)$. By the [Call] rule, $m = \mathsf{jsr}(\boldsymbol{\delta}(p))$ and $p \in \mathrm{dom}(\boldsymbol{\delta})$. A consequence of $\mathsf{fit}(\eta, \boldsymbol{\delta}, u \cdot m \cdot v)$ is the existence of some x and y such that $x \cdot \boldsymbol{\delta}(p) \cdot y = u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v$. As expected, the assumptions about the bindings lets us solve this case very elegantly:

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.(u, \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v, a) \\
\geq & \quad \{\text{Law [Jsr-wrp], } \text{“}\mathsf{StackOverflow}\text{”} \in A, \text{ existence of } x \text{ and } y\} \\
 & \mathsf{wrp}_A.(x, \boldsymbol{\delta}(p) \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v\rangle) \\
\geq & \quad \{\text{Second conjunct of } \mathsf{fit}(\eta, \boldsymbol{\delta}, u \cdot m \cdot v)\} \\
 & \eta(p) \,;\, \mathsf{wrp}_A.(u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v, a) \\
= & \quad \{\text{Semantics of a call}\} \\
 & \mathsf{wrp}^{\eta}_A.\mathsf{call}(p) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v, a) \;\; .
\end{array}
$$

Case f.) $\pi = \mathsf{blk}(p, \pi_p, \pi_b)$. By the [Blk] rule, there are assembly programs m_p ,m_b and labels l_p, l_b such that $m = \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b$ and $C(\pi_p, m_p, \boldsymbol{\delta} \cdot \{p \mapsto l_p\})$ and $C(\pi_b, m_b, \boldsymbol{\delta} \cdot \{p \mapsto l_p\})$ hold. Here we would like to calculate as follows:

$$
\begin{array}{ll}
 & \mathsf{wrp}_A.(u, \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, a) \\
\geq & \quad \{\text{Laws [Goto-wrp] and [Label-wrp]}\} \\
 & \mathsf{wrp}_A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b, m_b \cdot v, a) \\
\geq & \quad \{\text{Induction hypothesis: } C(\pi_b, m_b, \boldsymbol{\delta} \cdot \{p \mapsto l_p\}) \text{ holds}\} \\
 & \mathsf{wrp}^{\eta[p \mapsto \mu B_{p,\pi_p,A,\eta}]}_A.\pi_b \,;\, \mathsf{wrp}_A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \\
= & \quad \{\text{Semantics of blocks}\} \\
 & \mathsf{wrp}^{\eta}_A.\mathsf{blk}(p, \pi_p, \pi_b) \,;\, \mathsf{wrp}_A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \;\; .
\end{array}
$$

In order to apply the induction hypothesis in the second step, however, we have to check

$$
\mathsf{fit}(\eta[p \mapsto \mu B_{p,\pi_p,A,\eta}], \boldsymbol{\delta} \cdot \{p \mapsto l_p\}, u \cdot m \cdot v) \;\; .
$$

Unrolling the definition this means to prove

$$
\begin{array}{ll}
(*) & \exists x, y :: \\
 & \quad x \cdot \boldsymbol{\delta} \cdot \{p \mapsto l_p\}(q) \cdot y = u \cdot m \cdot v \;\wedge \\
 & \quad \forall e, f, g :: \; \mathsf{wrp}_A.(x, \boldsymbol{\delta} \cdot \{p \mapsto l_p\}(q) \cdot y, g \cdot \langle e, f\rangle) \\
 & \qquad\qquad \geq \eta[p \mapsto \mu B_{p,\pi_p,A,\eta}](q) \,;\, \mathsf{wrp}_A(e, f, g)
\end{array}
$$

for all $q \in \mathrm{dom}(\boldsymbol{\delta} \cdot \{p \mapsto l_p\})$. So suppose given an arbitrary $q \in \mathrm{dom}(\boldsymbol{\delta} \cdot \{p \mapsto l_p\})$. If $q \neq p$, $(*)$ reduces to

$$
\begin{array}{l}
\exists x, y :: \\
\quad x \cdot \boldsymbol{\delta}(q) \cdot y = u \cdot m \cdot v \;\; \wedge \\
\quad \forall e, f, g :: \; \mathsf{wrp}_A.(x, \boldsymbol{\delta}(q) \cdot y, g \cdot \langle e, f\rangle) \geq \eta(q) \,;\, \mathsf{wrp}_A.(e, f, g) \;\; ,
\end{array}
$$

which directly follows from the premise $\mathsf{fit}(\eta, \boldsymbol{\delta}, u \cdot m \cdot v)$. For $q = p$, on the other hand, we must prove

$$
\begin{array}{l}
\exists x, y :: \\
\quad x \cdot l_p \cdot y = u \cdot m \cdot v \;\wedge \\
\quad \forall e, f, g :: \; \mathsf{wrp}_A.(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq \mu B_{p,\pi_p,A,\eta} \,;\, \mathsf{wrp}_A.(e, f, g) \;\; .
\end{array}
$$

Choosing $x = u \cdot \mathsf{goto}(l_b)$ and $y = m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v$ makes the first conjunct true. The second conjunct is established by a fixpoint induction for $\mu B_{p,\pi_p,A,\eta}$.[6] Admissibility

[6] To be more precise, it is shown that $\mu B_{p,\pi_p,A,\eta} \in P_{x,l_p,y}$ where

is straightforward – follows from the distribution properties of the composition operator – and the base case easily follows from the fact that $\bot \,;\, \mathsf{wrp}_A(e,f,g) = \bot$. For the induction step assume that X is given such that for all e, f, g

$$(**) \quad \mathsf{wrp}_A.(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq X \,;\, \mathsf{wrp}_A(e,f,g) \ .$$

Now, $\mathsf{fit}(\eta[p \mapsto X], \boldsymbol{\delta} \cdot \{p \mapsto l_p\}, u \cdot m \cdot v)$ holds: for $q \neq p$ we can argue as above and for $q = p$ this follows from $(**)$. Thus, by using the induction hypothesis of the structural induction applied to π_p we can calculate as follows for arbitrarily given e, f, g:

$$\begin{array}{ll} & \mathsf{wrp}_A.(x, l_p \cdot y, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Law [Label-wrp] and unfolding of } y\} \\ & \mathsf{wrp}_A(x \cdot l_p, m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Induction hypothesis applied to } \pi_p\} \\ & \mathsf{wrp}_A^{\eta[p \mapsto X]}.\pi_p \,;\, \mathsf{wrp}_A.(x \cdot l_p \cdot m_p, \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Definition of } B_{p,\pi_p,A,\eta} \text{ and law [Ret-wrp]}\} \\ & B_{p,\pi_p,A,\eta}(X) \,;\, \mathsf{wrp}_A(e,f,g) \ , \end{array}$$

which completes the fixpoint induction, hence the proof for this case, altogether the proof of Lemma 8.2.1, and thus Theorem 8.2.1 by suitable instantiation. □

A variant of preservation of partial correctness. The theorem to be discussed and proved here reads of course quite similar.

Theorem 8.2.2. Suppose $\infty \in A$ and "StackOverflow" $\in A$. Then for all π and m:

$$\mathcal{C}(\pi, m, \emptyset) \ \Rightarrow \ \mathsf{wrp}_A.m \geq \mathsf{wrp}_A^{\bot_{Env}}.\pi \ .$$

Putting Theorem 8.2.2 into words: Computations of the target program starting in an initial state s either yield outcomes which are also possible for the source program, are outcomes to be accepted, or are ones for which the source program behaves chaotically if started in s, i.e. for which it does not satisfy its specification. To reword in the concrete setting, each regular result produced by π' is a possible result for π but π' is allowed to produce a "StackOverflow" or to deliver no result at all. In contrary to Theorem 8.2.1 a property like this is uninteresting for a process programmer as the target program may spontaneously diverge. To prove compilers correct, however, this is an extremely useful effect; as one typically wants to *use* the result of a compilation it suffices to get a result at all and it should be possible to force the compiler to produce a result at least once.

A look at the proof of Lemma 8.2.1 above unveils that we barely have a chance to perform a similar proof. The cause for this dilemma lies in the choice of fixpoints in the definition of the semantics for loops and blocks. As we accept divergence here, $\infty \in A$, the semantics of those commands is given in terms of greatest

$P_{x,l_p,y} \stackrel{\text{def}}{=} \{X \in PTrans \mid \forall e, f, g :: \ \mathsf{wrp}_A.(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq X \,;\, \mathsf{wrp}_A.(e,f,g)\}$.

$$
\begin{array}{ll}
\text{[Asg-D]} & D.f.A.(u, \mathsf{asg}(x,e)\cdot v, a) \\
& \quad \geq (x :=_A e)\ ;\ f.A.(u\cdot \mathsf{asg}(x,e), v, a) \\
\text{[Cj-D]} & D.f.A.(u, \mathsf{cj}(b,l)\cdot v, a) \\
& \quad \geq f.A.(u\cdot \mathsf{cj}(b,l), v, a) \lhd b/A \rhd f.A.(x, l\cdot y, a) \\
& \qquad \text{if } u\cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y \\
\text{[Goto-D]} & D.f.A.(u, \mathsf{goto}(l)\cdot v, a) \ \geq\ f.A.(x, l\cdot y, a) \\
& \qquad \text{if } u\cdot \mathsf{goto}(l)\cdot v = x\cdot l\cdot y \\
\text{[Jsr-D]} & D.f.A.(u, \mathsf{jsr}(l)\cdot v, a) \ \geq\ f.A.(x, l\cdot y, a\cdot\langle u\cdot \mathsf{jsr}(l), v\rangle) \\
& \qquad \text{if } u\cdot \mathsf{jsr}(l)\cdot v = x\cdot l\cdot y \text{ and } \text{“StackOverflow”} \in A \\
\text{[Ret-D]} & D.f.A.(u, \mathsf{ret}\cdot v, a\cdot\langle x, y\rangle) \ \geq\ f.A.(x, y, a) \\
\text{[Label-D]} & D.f.A.(u, l\cdot v, a) \ \geq\ f.A.(u\cdot l, v, a) \\
\text{[Term-D]} & D.f.A.(u, \varepsilon, a) \ \geq\ Id
\end{array}
$$

Table 8.2. 'D'-laws for the assembly language.

fixpoints and to apply the induction rule, the transfer lemma or to perform a pure fixpoint induction – as done in the while- and blk-cases before – the fixpoint is on the wrong side of the order symbol.

Since we cannot get rid of the mentioned fixpoints we have to go a completely different way. The alternative approach performed below is motivated by the skin deep insight that it is easy to show that something is greater (w.r.t. '$\geq$') than the smallest fixpoint but that it is hardly possible to show that something is greater than the greatest fixpoint. Formulated the other way round it should be easy to show that something is smaller than the greatest fixpoint but the question remains to which fixpoint this statement could refer. Luckily, we already know the answer: Sect. 7.1 presents a fixpoint characterization of wrp-transformers for languages whose semantics is given by means of a purely operational semantics. Thus, the scenario presented there is applicable for the target language and, in fact, we already exploited this characterization in Sect. 7.3 where we derived rules for the assembly language.

The idea is to reason in terms of function D as defined by (7.4) such that, by Theorem 7.1.2, showing $\mathsf{wrp}_A.m \geq \mathsf{wrp}_A^{\perp_{Env}}.\pi$ as required in Theorem 8.2.2 amounts to showing $\nu D.A.m \geq \mathsf{wrp}_A^{\perp_{Env}}.\pi$, and the latter presentation suggests to perform a fixpoint induction for νD. In our concrete setting function D enjoys the properties collected in Tab. 8.2 which look quite similar to the laws presented in Tab. 7.2 and so do the proofs.

This view on wrp-transformers allows to approach the claim at all but the problems do not run out. In the present scenario we also have to quantify over all contexts in order to bridge the compositionality gap. Obviously, reasonable relations between the semantical bindings in the environment and the syntactical bindings in the used translation dictionary have to be expressed, too. It turns out that the nice and rather simple fit predicate, as defined by (8.5), or modifications thereof do not work. The reason is already known from Sect. 7.4 when the equivalence of the operational and the denotational semantics of the source language was shown: One has to express that old bindings do not change. The remedy

there was the particular property of the operational semantics that it deals with fresh identifiers and distinguished procedure dictionaries which allowed to rename programs and to upgrade environments with fresh identifiers, cf. Theorem 7.4.1 and Lemma 7.4.3. However, to ensure that the bindings are not violated it is also helpful to have direct access to them, and this from the view of the source and the target language. Thus, for the actual proof, let us assume that besides the stack of dictionaries used for the translation we also have a stack $\boldsymbol{\eta}$ of environments available. (We adopt the notational conventions from stacks of dictionaries and particularly assume that the topmost entry of a stack of environments $\boldsymbol{\eta}$ is η.) Then the predicate below expresses what can be guaranteed here and it will support the oncoming proof a lot. The chosen name is inspired by the visual impression that it seems to be somehow stronger that the fit-predicate, this might indeed be the case but we refrain from an attempt to prove this.

$$
\begin{array}{l}
\mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u) \quad \stackrel{\text{def}}{\Longleftrightarrow} \\
\quad |\boldsymbol{\eta}| = |\boldsymbol{\delta}| \ \ \wedge \\
\quad \forall \boldsymbol{\delta}_i \in \boldsymbol{\delta} :: \\
\qquad \forall q \in \mathrm{dom}(\boldsymbol{\delta_i}) :: \ \exists x, y, m_q, \pi_q :: \\
\qquad\quad x \cdot \boldsymbol{\delta_i}(q) \cdot m_q \cdot \mathsf{ret} \cdot y = u \ \ \wedge \\
\qquad\quad \mathcal{C}(\pi_q, m_q, \boldsymbol{\delta_{j_q}}) \ \ \wedge \\
\qquad\quad \mathsf{wrp}_A^{\eta_{j_q}}.\pi_q \geq \eta_i(q) \ \ .
\end{array}
\tag{8.6}
$$

Verbalizing $\mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u)$, for each prefix-stack $\boldsymbol{\delta_i}$ the following holds. If a procedure q has been declared so far, i.e $q \in \boldsymbol{\delta_i}$, then a corresponding label, $\boldsymbol{\delta_i}(q)$, followed by a procedure body, m_q, and a return command together with a surrounding context can be found. This procedure body, m_q, is the result of a translation of a procedure body, π_q, where dictionary $\boldsymbol{\delta_{j_q}}$ was used which was the current at that time ($\boldsymbol{\delta_{j_q}}$ must be a prefix of $\boldsymbol{\delta_i}$ which is reasonable because $q \in \boldsymbol{\delta_i}$). Furthermore, the semantics of π_q was given by means of a wrp-transformer using an environment that was the current at that time, too. As j_q is the index where q was introduced lastly the current environment, i.e. η_i, applied to q should not have changed either. This expresses in the last conjunct of fitter.

As the stacks of environments and dictionaries are assumed to be of equal length and because each prefix is described in isolation the fitter-predicate enjoys the following and barely surprising prefix-property, the easy proof of which is left to the reader.

Lemma 8.2.2 (Prefix property). If $|\boldsymbol{\eta}| = |\boldsymbol{\delta}|$ and $|\boldsymbol{\eta'}| = |\boldsymbol{\delta'}|$ then

$$\mathsf{fitter}(\boldsymbol{\eta} \cdot \boldsymbol{\eta'}, \boldsymbol{\delta} \cdot \boldsymbol{\delta'}, u) \implies \mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u) \ \ .$$

□

Similar to Lemmas 7.4.5 and 7.4.7 it can also be easily initialized by choosing the singleton stacks consisting of the "everywhere undefined" dictionary $\emptyset$ resp. the "constantly false" environment $\bot_{Env}$.

Now, the needed preparations are made to state the appropriate claim for the variant of preservation of partial correctness.

Lemma 8.2.3. Suppose $\infty \in A$ and "StackOverflow" $\in A$. For all π, m, u, v ,a, $\boldsymbol{\eta}$, $\boldsymbol{\delta}$:

$$\begin{aligned}&\mathcal{C}(\pi, m, \boldsymbol{\delta}) \wedge \mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot m \cdot v) \\ &\quad \Rightarrow \mathsf{wrp}_A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; \mathsf{wrp}_A.(u \cdot m, v, a) \;\;.\end{aligned} \tag{8.7}$$

Again, Theorem 8.2.2 follows by suitable instantiation: $u = v = \varepsilon$, $a = \varepsilon$, $\eta = \bot_{Env}$ and $\boldsymbol{\delta} = \emptyset$.

Proof of Lemma 8.2.3. As motivated before we perform a fixpoint induction for νD such that, in the very end, we obtain

$$\nu D.A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; \nu D.A.(u \cdot m, v, a)$$

which is equivalent to the claim of (8.7) by Theorem 7.1.2. Admissibility is straightforward – here, this follows from wrp_A^{π} being conjunctive – and so is the base case; the adequate transformer is the $\top$-transformer of the considered lattice which constantly yields `true` for all its arguments. For the induction step we assume given a transformer f with $f \geq D.f$ satisfying the fixpoint induction hypothesis, i.e.

$$\begin{aligned}&\mathcal{C}(\pi, m, \boldsymbol{\delta}) \wedge \mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot m \cdot v) \\ &\quad \Rightarrow f.A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; f.A.(u \cdot m, v, a)\end{aligned}$$

for all π, m, u, v ,a, $\boldsymbol{\eta}$ and $\boldsymbol{\delta}$. As usual we proceed by a structural induction for the actual fixpoint induction step which means to show

$$\begin{aligned}&\mathcal{C}(\pi, m, \boldsymbol{\delta}) \wedge \mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot m \cdot v) \\ &\quad \Rightarrow D.f.A.(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}.\pi \;;\; D.f.A.(u \cdot m, v, a)\end{aligned}$$

for each π in turn. We thus assume the premise to hold for the program in question and also the entire claim for the structural components of which it consists.

Case a.) $\pi = \mathsf{assign}(x, e)$. By the [Assign] compiling rule, $m = \mathsf{asg}(x, e)$ and

$$\begin{aligned}&D.f.A.(u, \mathsf{asg}(x, e) \cdot v, a) \\ \geq \quad &\{\text{Law [Asg-D]}\} \\ &(x :=_A e) \;;\; f.A.(u \cdot \mathsf{asg}(x, e), v, a) \\ \geq \quad &\{\text{Semantics of assignments and } f \geq D.f\} \\ &\mathsf{wrp}_A^{\eta}.\mathsf{assign}(x, e) \;;\; D.f.A.(u \cdot \mathsf{asg}(x, e), v, a) \;\;.\end{aligned}$$

Case b.) $\pi = \mathsf{seq}(\pi_1, \pi_2)$. By the [Seq] compiling rule, there are m_1 ,m_2 with $m = m_1 \cdot m_2$ with $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$ and $\mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})$. Here,

$$\begin{aligned}&D.f.A.(u, m_1 \cdot m_2 \cdot v, a) \\ \geq \quad &\{\text{Induction hypothesis for } \pi_1\} \\ &\mathsf{wrp}_A^{\eta}.\pi_1 \;;\; D.f.A.(u \cdot m_1, m_2 \cdot v, a) \\ \geq \quad &\{\text{Induction hypothesis for } \pi_2\}\end{aligned}$$

$$
\begin{array}{ll}
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \,;\, \mathsf{wrp}^{\eta}_{A}.\pi_2 \,;\, D.f.A.(u \cdot m_1 \cdot m_2, v, a) \\
= & \qquad \{\text{Semantics of composition}\} \\
& \mathsf{wrp}^{\eta}_{A}.\mathsf{seq}(\pi_1, \pi_2) \,;\, D.f.A.(u \cdot m_1 \cdot m_2, v, a) \ .
\end{array}
$$

Case c.) $\pi = \mathsf{if}(b, \pi_1, \pi_2)$. By the [If] compiling rule, $m = \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2$ for some m_1, m_2 with $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$ and $\mathcal{C}(\pi_2, m_2, \boldsymbol{\delta})$. In this case

$$
\begin{array}{ll}
& D.f.A.(u, \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a) \\
\geq & \qquad \{\text{Law [Cj-D]}\} \\
& f.A.(u \cdot \mathsf{cj}(b, l_1), m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2), l_1 \cdot m_2 \cdot l_2 \cdot v, a) \\
\geq & \qquad \{f \geq D.f \text{ and induction hypothesis for } \pi_1 \text{ in the first,} \\
& \qquad\ \text{Law [Label-D] in the second component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \,;\, D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1, \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2 \cdot v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1, m_2 \cdot l_2 \cdot v, a) \\
\geq & \qquad \{\text{Law [Goto-D] in the first component,} \\
& \qquad\ f \geq D.f \text{ and induction hypothesis for } \pi_2 \text{ in the second}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \,;\, f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2, l_2 \cdot v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}^{\eta}_{A}.\pi_2 \,;\, D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2, l_2 \cdot v, a) \\
\geq & \qquad \{f \geq D.f \text{ in the first component, then law [Label-D]} \\
& \qquad\ \text{in both followed by } f \geq D.f \text{ again}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \,;\, D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
& \lhd b/A \rhd \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \,;\, D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
= & \qquad \{\text{Distribute conditional, i.e. Lemma 7.2.1}\} \\
& (\mathsf{wrp}^{\eta}_{A}.\pi_1 \lhd b/A \rhd \mathsf{wrp}^{\eta}_{A}.\pi_1) \,; \\
& D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \\
= & \qquad \{\text{Semantics of conditional}\} \\
& \mathsf{wrp}^{\eta}_{A}.\mathsf{if}(b, \pi_1, \pi_2) \,;\, D.f.A.(u \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, v, a) \ .
\end{array}
$$

Case d.) $\pi = \mathsf{while}(b, \pi_1)$. By the [While] compiling rule, $m = l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1$ where m_1 is the one with $\mathcal{C}(\pi_1, m_1, \boldsymbol{\delta})$. The calculation below heavily depends on the hypothesis of the fixpoint induction:

$$
\begin{array}{ll}
& D.f.A.(u, l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
\geq & \qquad \{\text{Law [Label-D], then } f \geq D.f \text{ and law [Cj-D]}\}
\end{array}
$$

$$
\begin{array}{ll}
& f.A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1), m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0), l_1 \cdot v, a) \\
\geq & \quad \{f \geq D.f \text{ and induction hyposthesis for } \pi_1 \text{ in the first,} \\
& \quad\ \text{law [Label-D] in the second component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; D.f.A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1, \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a) \\
\geq & \quad \{\text{Law [Goto-D] in the first component}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; f.A.(u, l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \cdot v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1, v, a) \\
\geq & \quad \{\text{Apply the fixpoint-induction hypothesis to} \\
& \quad\ m = l_0 \cdot \mathsf{cj}(b, l_1) \cdot m_1 \cdot \mathsf{goto}(l_0) \cdot l_1 \text{ in the first component!}\} \\
& \mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b, \pi_1) \ ; f.A.(u \cdot m, v, a) \\
& \lhd b/A \rhd \\
& f.A.(u \cdot m, v, a) \\
\geq & \quad \{\text{Distribute conditional and } f \geq D.f\} \\
& (\mathsf{wrp}^{\eta}_{A}.\pi_1 \ ; \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b, \pi_1) \lhd b/A \rhd Id) \ ; D.f.A.(u \cdot m, v, a) \\
= & \quad \{\text{Definition of } \mathcal{W}_{b, \mathsf{wrp}^{\eta}_{A}.\pi_1}\} \\
& \mathcal{W}_{b, \mathsf{wrp}^{\eta}_{A}.\pi_1}(\mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b, \pi_1)) \ ; D.f.A.(u \cdot m, v, a) \\
= & \quad \{\text{Definition of } \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b, \pi_1)\} \\
& \mathcal{W}_{b, \mathsf{wrp}^{\eta}_{A}.\pi_1}(\nu \mathcal{W}_{b, \mathsf{wrp}^{\eta}_{A}.\pi_1}) \ ; D.f.A.(u \cdot m, v, a) \\
= & \quad \{\text{Unroll fixpoint}\} \\
& \nu \mathcal{W}_{b, \mathsf{wrp}^{\eta}_{A}.\pi_1} \ ; D.f.A.(u \cdot m, v, a) \\
= & \quad \{\text{Semantics of loops again}\} \\
& \mathsf{wrp}^{\eta}_{A}.\mathsf{while}(b, \pi_1) \ ; D.f.A.(u \cdot m, v, a) \ .
\end{array}
$$

Case e.) $\pi = \mathsf{call}(p)$. By the [Call] rule, $m = \mathsf{jsr}(\boldsymbol{\delta}(p))$ and $p \in \mathrm{dom}(\boldsymbol{\delta})$. The premise, i.e. $\mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v)$ in this case, lets us conclude the existence of some x, y, m_p and π_p such that

1.) $x \cdot \boldsymbol{\delta}(p) \cdot m_p \cdot \mathsf{ret} \cdot y \ = \ u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v$,

2.) $\mathcal{C}(\pi_p, m_p, \boldsymbol{\delta_{j_p}})$, and

3.) $\mathsf{wrp}^{\eta_{j_p}}_{A}.\pi_p \geq \eta(p)$.

A consequence of the "prefix property", i.e. Lemma 8.2.2, is furthermore that

4.) $\mathsf{fitter}(\boldsymbol{\eta_{j_p}}, \boldsymbol{\delta_{j_p}}, u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v)$.

These observations allow to calculate as follows:

$$
\begin{array}{ll}
& D.f.A.(u, \mathsf{jsr}(\boldsymbol{\delta}(p)) \cdot v, a) \\
= & \quad \{\text{Law [Jsr-D] and 1.), note that } \boldsymbol{\delta}(p) \text{ is unique}\} \\
& f.A.(x, \boldsymbol{\delta}(p) \cdot m_p \cdot \mathsf{ret} \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v\rangle) \\
\geq & \quad \{f \geq D.f \text{ and law [Label-D]}\} \\
& f.A.(x \cdot \boldsymbol{\delta}(p), m_p \cdot \mathsf{ret} \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v\rangle) \\
\geq & \quad \{\text{Apply the fixpoint induction hypothesis to } \pi_p, \\
& \quad\ \text{this is admissible by 2.) and 4.)!}\} \\
& \mathsf{wrp}_A^{\eta_{j_p}}.\pi_p \ ; \ f.A.(x \cdot \boldsymbol{\delta}(p) \cdot m_p, \mathsf{ret} \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v\rangle) \\
\geq & \quad \{f \geq D.f \text{ and law [Ret-D]}\} \\
& \mathsf{wrp}_A^{\eta_{j_p}}.\pi_p \ ; \ f.A.(u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v, a) \\
\geq & \quad \{\text{See 3.)}\} \\
& \eta(p) \ ; \ f.A.(u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v, a) \\
\geq & \quad \{f \geq D.f \text{ and semantics of a call}\} \\
& \mathsf{wrp}_A^{\eta}.\mathsf{call}(p) \ ; \ D.f.A.(u \cdot \mathsf{jsr}(\boldsymbol{\delta}(p)), v, a) \ .
\end{array}
$$

Case f.) $\pi = \mathsf{blk}(p, \pi_p, \pi_b)$. By the [Blk] rule, there are assembly programs m_p, m_b and labels l_p, l_b with

1.) $C(\pi_p, m_p, \boldsymbol{\delta} \cdot \{p \mapsto l_p\})$ and
2.) $C(\pi_b, m_b, \boldsymbol{\delta} \cdot \{p \mapsto l_p\})$,

such that $m = \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b$. Here, we like to apply the hypothesis of the structural induction to the body π_b. (We could also use the fixpoint induction hypothesis but we refrain from doing so as the translation scheme could be modified in such ways that it delivers $m_b \cdot \mathsf{goto}(l) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l$. In this case the fixpoint induction hypothesis does not apply and we like to be as general as possible.) In order to do so we have to establish

3.) $\mathsf{fitter}(\boldsymbol{\eta} \cdot \eta[p \mapsto \nu B_{p.\pi_p,A,\eta}], \boldsymbol{\delta} \cdot \{p \mapsto l_p\}, u \cdot m \cdot v)$,

so let us start with this. It suffices to consider the entire stack because for all prefixes the claim follows from the premise $\mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot m \cdot v)$. So assume given an arbitrary $q \in \mathrm{dom}(\boldsymbol{\delta} \cdot \{p \mapsto l_p\})$. If $q \neq p$ we are done because $\boldsymbol{\delta} \cdot \{p \mapsto l_p\}(q) = \boldsymbol{\delta}(q)$ and likewise $\eta[p \mapsto \nu B_{p.\pi_p,A,\eta}](q) = \eta(q)$ so that with $j_q < |\boldsymbol{\delta}| + 1$ the required properties follow already from $\mathsf{fitter}(\boldsymbol{\eta}, \boldsymbol{\delta}, u \cdot m \cdot v)$ again. So assume $p = q$. We already know m_p and π_p so choosing $x = u \cdot \mathsf{goto}(l_b)$ and $y = l_b \cdot m_b \cdot v$ makes the first conjunct true. As $j_q = |\boldsymbol{\delta}| + 1$ in this case we also have the second conjunct by 1.) above. To prove the third conjunct we calculate

$$
\begin{array}{ll}
& \eta[p \mapsto \nu B_{p.\pi_p,A,\eta}](p) \\
= & \quad \{\text{Application}\} \\
& \nu B_{p.\pi_p,A,\eta} \\
= & \quad \{\text{Unroll the fixpoint and definition of } B\} \\
& \mathsf{wrp}_A^{\eta[p \mapsto \nu B_{p.\pi_p,A,\eta}]}.\pi_p \ ,
\end{array}
$$

such that, all in all, 3.) indeed holds. Finally, this allows to calculate as intended:

$$
\begin{array}{ll}
 & D.f.A.(u, \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, a) \\
\geq & \quad \{\text{Law [Goto-D]}\} \\
 & f.A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret}, l_b \cdot m_b \cdot v, a) \\
\geq & \quad \{f \geq D.f \text{ and law [Label-D] and } f \geq D.f \text{ again}\} \\
 & D.f.A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b, m_b \cdot v, a) \\
\geq & \quad \{\text{Apply structural induction hypothesis to } \pi_b, \\
 & \quad\ \text{this is admissible by 2.) and 3.)!}\} \\
 & \mathsf{wrp}_A^{\eta[p \mapsto \nu B_{p.\pi_p, A, \eta}]}.\pi_b \ ; D.f.A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \\
= & \quad \{\text{Semantics of blocks}\} \\
 & \mathsf{wrp}_A^{\eta}.\mathsf{blk}(p, \pi_p, \pi_b) \ ; D.f.A.(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \ ,
\end{array}
$$

and this completes the structural induction, by this the fixpoint induction step, altogether the proof of Lemma 8.2.3 and ultimatively Theorem 8.2.2.
□

We close this section with the observation that the compiling specification preserves the termination behavior up to the occurrance of "StackOverflow"-errors, i.e. regular terminating resp. diverging source programs are translated to regular terminating resp. diverging target programs unless the executing machine reports a "StackOverflow"-error. Note that we can conclude this preservation of the termination behavior without any termination proofs. This result is more precisely given in the final theorem below which is a direct application of Theorem 4.3.2.

Theorem 8.2.3 (A correct implementation). Suppose $\mathcal{C}(\pi, m, \emptyset)$. Then m implements π with preserved outcomes $PO = \Sigma \cup \{\infty\}$, accepted outcomes $AO = \{\text{"StackOverflow"}\}$ and chaotic outcomes $CO = (\Sigma \cup \Omega) \setminus (PO \cup AO)$.
□

8.3 Remarks

A precise analysis of optimizations is a nice application of the relativized scenario. Similar looking (!) calculations on an even more abstract and thus more readable level could have been performed in the refinement calculus (which corresponds to preservation of total correctness) or similar frameworks but it is very noteworthy that the results would have been quite different. The reason is always the same: Typically the presence of so-called finite errors is disregarded in the area of predicate transformer semantics. Let us just mention that dead code elimination and unswitching preserve both partial and total correctness if one desists from finite errors though the former does not preserve partial and the latter does not preserve total correctness in the "real world" as seen in Theorem 8.1.1 resp. 8.1.4. Furthermore, as announced, wrp-based reasoning indeed allows to distinguish between different erroneous outcomes and thus to state precise requirements which

have to be satisfied in order to establish a specific relative correctness property. To underpin these remarks we advise to have a look at [41] which proves optimizations correct in the framework of a so-called "Kleene algebra with tests" which is a Kleene algebra (the algebra of regular expressions) with an embedded Boolean Algebra. The operators thereof are assumed to model some typical programming operators and thus correct program transformations resp. optimizations correspond to valid laws in the algebra. Due to the calculational style the presented proofs are quite elegant and short but, as usual, the results do not transfer to reality because the presence of finite errors and even divergence is not taken into account.

Remarkable in this connection is the particular feature of relative correctness and the way it is dealt with, i.e. the calculational style using wrp-transformers, that these searched requirements intuitively pop up while attempting the proof. This is the case for all demands on outcomes presented here; in particular the observation that unswitching does not preserve total correctness and also the cause for it was found in the actual proof and not a pretended requirement which was pushed into it.

In Sect. 8.1 we acquired the used vocabulary and concerning laws when they were needed, and this in a form that suits the particular task. Since the present thesis is meant for a general introduction to relative correctness and its application this is an appropriate proceeding. But as the relativized setup has demonstrated its utility – at least this is our opinion – it might be suggestive to elaborate a construction kit of wrp-rules, similar to the refinement calculus, e.g. [53], or [61] and far beyond the very basic rules presented in Chap. 6 and Sect. 8.1, which facilitates point-free reasoning about realistic program transformations on an accurate and handy level.

On the other hand, proving correct translations of toy-languages – in a comprehensible algebraic manner – is a seemingly old-fashioned exercise. Nevertheless there is a great need for translation verification and both the languages and, moreover, the correctness proofs presented here are very worth mentioning for the following reasons.

Firstly, and we already discussed this at length, procedures are included in both the source and the target language. Though only parameterless procedures are considered the control flow aspects are quite more exciting and the languages are generally more common in the sense that they are an adequate means to study, apart from data flow aspects, the essence of ALGOL-like resp. assembly languages. The only similar approach we know of is [47] which views on "compilation as refinement" (see also [46]). The source language considered there knows of procedures, too, and the semantics of both the source and target language is given by means of a common modeling language which is basically the one of the refinement calculus as presented in [53]. In contrast to our proceeding no translation scheme is given which is proved (post mortem) to have a specific preservation property, instead source programs are inductively "flattened", i.e. replaced by sequences of instructions of the target language, by application of refinement rules. In so doing – this is the claim – the translation is correct by definition because

the rules of the refinement calculus are correct. However, we dare to say that certain aspects of this approach remain to be discussed. The source language allows nesting of procedures but this phenomenon is not mentioned in [47] at all. The semantics of procedure calls is given by application of a naive copy rule, i.e. simple replacement of a call by its body (as defined in [53]) which corresponds to dynamic scoping, but for nested procedures one has to be rather careful to which body a call refers in order to model static scoping (if desired). The translation, i.e. the specific "flattening", of procedure bodies and calls on the other hand seems to implement static scoping so the (correctness of the) translation remains questionable anyhow. Moreover, it can hardly be reconstructed because the interesting parts of the proof are omitted. Furthermore, since finite errors are not included in the model and classical refinement is the used medium, the approach deals with preservation of total correctness only.

Secondly, programs are assumed to be executed on finite machines. This is a vision close to reality but it is also a presentation out of the ordinary what more abstract and comprehensible scenarios concerns. Though one should be conscious that resource limitations do exist in reality one typically ignores this fact (for convenience and simplicity) or focuses on avoiding finite errors; consequently total correctness and preservation thereof is the notion which is of interest mostly. Remarkably here is, e.g., [30] which provides a (relational) framework of specifications in which time- and space-bounds can be taken into account by additional specification variables that are outside the actual state space. By this means it is at least possible to reason about resource limitations even if they are actually not modeled in the semantics. However, having a slightly more realistic semantics at hand it is natural to discuss also other preservation properties. As the target machine might report a "StackOverflow" on executing a procedure call, preservation of total correctness cannot be achieved. Since non-diverging source programs are not translated to diverging target programs this particular preservation property is also of great value and interest but, unfortunately, the notion of preservation of partial correctness cannot distinguish divergence from aborts. We hope the reader appreciates that our wrp-approach is – on contrary to the classical setup – indeed able to get along with this as it does not mix divergence with finite errors. Nevertheless, preservation of partial correctness and variations thereof are very worth mentioning because they are of practical interest if compilers are to be proved correct. We know of no contributions – apart from ones by the *Verifix* project (e.g. [23, 25]) in which the present thesis has its roots – that pursue properties like these. In this sense the second translation correctness proof given here is rather new, barely known and truly worth reading.

An interesting aspect of our proof is that it shows how to handle the transition from tree-structured source programs to "flat" target code. For this purpose we established a stronger claim that involves a universal quantification over syntactic target program contexts. This should be contrasted to the use of tree-structured assembler languages, e.g. in [65], where translation correctness for a WHILE-language without procedures is investigated. The proof in [65] does not immediately generalize to flat code.

A widely asked question shall be discussed now: What is the harder task, proving PTC or PPC? A look at the proofs unveils that the second part is the seemingly harder one. A structural induction does not suffice and apply, instead a fixpoint induction has to be performed. Furthermore an apparently stronger predicate has to be used in order to express sensible relationships between the bindings of translation dictionaries and environments. The most intuitive cause for this observation is the following. A structural induction allows to infer properties of a compound from properties of structurally smaller components. This is obviously the case for the syntax of the high-level language considered here but in the end we are not interested in syntax but in semantics. If the semantics confirms – in some structural sense – with the syntax then structural induction seems to suffice. Consider, for instance, the PTC case: For each syntactical component a *terminating* computation is assumed and thus the computation of a compound is pieced together of smaller (w.r.t. the computation length) computations. This is not the case for PPC. Here, each component may run forever and thus a compound may not be greater (w.r.t. the computation length) than the parts of which it consists. Besides this picture there is a strictly technical reason. Refinement is expressed by means of an inequality sign where the semantics of the source language, i.e. the structuring one in this case, is on the "less than or equal" side. As for PPC the semantics is defined via greatest fixpoints a structural induction amounts to showing that the semantic of the target language is greater than or equal to a greatest fixpoint (w.r.t. '$\geq$'), see also the comments made on p. 159, and common fixpoint rules simply do not apply.

However, it might also be kept for questionable whether PPC is really harder than PTC. As just motivated a structural induction does not suffice but notice the premise that the language is structured at all. The semantics of our high-level language was initially given operationally and it was a good piece of craftsmanship to show the equivalence of the operational and the denotational wrp-semantics, cf. Sect. 7.4. The proofs of Lemmas 7.4.14 and 7.4.15 seem to be the harder tasks because of the plenty of preparations but note that these are exactly the cases where the fixpoints of the source language are on the "wrong" side to apply nice fixpoint rules. If the semantics were to be compared on the operational level both proofs, the one for PTC and the one for PPC, might not differ that much because none of the involved semantics would be really structured. Furthermore, exaggerating a bit, performing the equivalence proof *together* with the translation proof is sort of zero-sum situation because for PTC the equivalence proof might be kept for marginally harder and the translation proof seems to be easier; for PPC the situation is conversely.

To mention yet another position: The proofs for PTC and PPC might be kept for not that different as they appear at first sight but just for turned inside out. For PTC a structural induction is performed and some cases of the induction step involve some fixpoint reasoning, including a fixpoint induction. For PPC, again, the situation is vice versa. A fixpoint induction is performed and the fixpoint induction step consists of a structural induction (without any further tricky fixpoint reasoning). Moreover, in the parlance of category theory and, e.g., [64] the

PPC case corresponds to a coinduction and the PTC case to an induction, see the comments made on p. 131. However, to underpin this view we like to stress the following. In the fixpoint induction step of the PPC case one makes heavy use of both the fixpoint and the structural hypothesis, and both hypotheses are really essential. At first sight, this seems to be different for PTC; the fixpoint reasoning inside the induction step of the structural induction seems to be independent of the structural hypothesis (in particular for the sub-language without procedures). But note that there are two, nearly hidden, applications of structural hypotheses inside the fixpoint reasoning. First of all, in the while case, an inequality was shown with aid of the structural hypothesis, and it gave rise to apply the induction rule resp. the transfer lemma. Secondly, a precise look at the actual fixpoint induction step of the blk-case, see p. 157, unveils that the structural hypothesis is also indispensable. (We like to mention that, in principle, the least fixpoint is on the "right" side to let the fixpoint induction rule apply – after shunting the right hand side to the left, note that function $(; R)$ is universally disjunctive, in order to isolate the fixpoint – but this will not lead to the goal for the following reasons. Roughly speaking, after some more shunting one is in need to apply the induction hypothesis to the procedure body under yet some other assumptions concerning the bindings. These assumptions are parameterized with the specific context e, f, g from where the calculation starts, but to establish these assumptions one is faced with showing the second conjunct of fit for all contexts, say, a, b, c, even for those different from the specific e, f, g, and this cannot be achieved in general. For a better understanding the interested reader is invited to figure this out. Note that pure fixpoint induction on the other hand allows a restriction to certain predicate transformers, namely to those which satisfy the premise for all contexts; see the footnote on p. 157.) Thus, even a very precise look affirms the statements made before: The used proof principles for PTC and PPC are just turned inside out.

As the case may be, we do not dare to decide which proof *is really* the harder one. We agree that the PTC proof is the more intuitive one and thus the seemingly easier exercise – in fact, it *was* the easier one and we do not know why the PPC case was such a crux – but having found a solution for PPC the latter looks quite elegant, comprehensible and in some sense even clearer from the operational perspective. Anyhow, it might only be a matter of taste.

9. Conclusion

Now, at the end of tour, we hope the reader appreciates and agrees with us that the notions of a correct implementation and relative correctness as well as weakest relative preconditions and particularly wrp-transformers are an adequate and manageable means for reasoning about more realistic translation verification exercises. They indeed balance the gap between approved theory and practical needs because as much as possible is preserved from the elegant appearance of the classical and idealized setting while being able to cope with the more authentic demands as well. To refresh the reader's mind let us briefly resume the major issues.

Mostly inspired by the skin deep insight that no, say, commercial compiler can ever be "correct" in any of the classical senses we proposed to have a more shrewd but still abstract view on different erroneous outcomes. A clever combination of this more careful distinction and the known theory of weakest precondition semantics promises to get along with modern, i.e. optimizing, compilers generating executables running on real machines, and this in a comprehensible and abstract manner.

Partitioning the set of all outcomes into ones to preserve literally, ones to accept and ones to reject, i.e. Def. 2.3.1, serves the first part of the proposal. It allows to express very precise and detailed demands on the compiler. Each translation task can be suited to its very particular field of application, be it for generating target programs which have to satisfy some safety and/or liveness properties, for constructing verified compilers where correctness of the result is more important than regular termination, where, e.g., a clean debugging feature is desired or not, where brute optimizations are allowed or not etc.

Def. 2.3.1 serves the first part of the proposal but the substantial benefit elaborated in these concerns is the more abstract and thus manageable treatment of this notion. Though it is advisable to keep an operational or relational semantics for the ultimative reference it is a thankless exercise to prove translations correct in the sense of Def. 2.3.1 directly. Motivated by the classical setup we defined a predicate transformer along the lines of Dijkstra's well-known wp and wlp but which does not mix divergence with aborts and which does not reject resp. tolerate any sort of irregular outcomes but only some, the irregular outcomes that are not accepted are taken as disproof. The notion of relative correctness and particularly the family of wrp_A-transformers serves the second part of our proposal. The greater selectivity, the more subtle differentiation between different outcomes and their causes, allows to express and reason about fine-grained correctness properties in

the sense of Def. 2.3.1 but on an abstract level and thus in more manageable and handy ways, see for instance Theorem 4.3.1.

On account of being slightly more realistic as well as supporting the compiler builder's need for modularity we also visited the scenario with inhomogeneous state spaces. For a brief introduction it is sufficient and recommendable to assume source and target programs to operate on a common state space but this is obviously not what the real world looks like. Data representation Galois connections turned out to be a nice and handy means for relating different state spaces, and with the aid of such it is straightforward to extend the notion of preservation of relative correctness to the inhomogeneous setting; in fact, the concept of correct implementations in the sense of Def. 2.3.1 transfers, too. Yet another advantage of relative correctness and wrp-based reasoning is that it supports the need to build businesslike compilers in a modular fashion. The more abstract view unveils that relative correctness is a transitive notion, and this even in very queer situations that might arise in reality.

The wrp-transformers were defined around the idea of Dijkstra's wp- and wlp-transformers. In contrast to our picture – we keep an underlying operational or relational semantics for the ultimative reference and thus the wrp-transformers are derived terms – those transformers are defined axiomatically, are postulated terms in [18]. Not surprisingly, our transformers meet Dijkstra's healthiness conditions, but even more. The basic laws concerning wrp are just more general versions of the axioms presented in [18], and it is this particular collection of laws that restricts the space of monotonic predicate transformers consistently to those which correspond to a relational semantics. To be precise, it is allowed to switch between the "full" relational semantics and wrp-transformers without any loss of information and this particular issue debilitates the prejudice that a predicate transformer semantics is too abstract for realistic verification purposes.

To embrace the title and to substantiate oncoming applications, the wrp-semantics of two common programming languages were introduced. The abstract assembly language is kept for a representive for a variety of flat assembly languages and it has the particular property that its semantics allows finite errors to emerge. It is equipped with the customary control flow operations which are assumed to be "macros" consisting of (sequences of) real machine-instructions; the appendix justifies our understanding of the language being *almost* an abstract view on existing languages. The high-level language serves as an adequate means to study the control flow aspects of ALGOL-like languages. Once more we like to stress that it is rather unusual to consider (nested) procedures in more abstract settings like the present. Though proving the equivalence of the operational and the denotational semantics was a hard exercise it is nice to see that procedures do not complicate reasoning about programs.

Finally, we showed that relative correctness and specifically wrp-based reasoning indeed keeps the promise to facilitate proving realistic translations running on real machines correct. As optimizations are ubiquitous in contemporary compilers and because optimizations gave rise to see that the classic correctness notions are not adequate in reality, we visited some strategies by example. It is very notewor-

thy that only relative correctness is able to cope with questions like these: Partial and total correctness really do not reflect reality in these concerns. The actual translation correctness proof provides a general proceeding to prove correct the transition from structured to flat languages, and this for various correctness notions that might be of interest. The most distinguishing aspect of correctness is the acceptance of divergence. Therefore, two correctness proofs are given which apply depending on whether $\infty \in A$ or not. The only finite error which reflects a violation of some resource limitations is the "StackOverflow"-error but it is straightforward to integrate others. Remarkable in this connection is that most of the proof is reusable in the sense that those parts of the structural induction which are independent of other errors transfer to the new situation. Moreover, the proof presented here is intended to serve as a guiding standard for even more complicated languages.

9.1 Topics for Future Research

A vast number of pitfalls and questions popped up while working on this thesis. Some approaches and solutions have been proposed but there is still a good deal of further work.

Case Study. The initial and actual task stems from the *Verifix* project: Prove preservation of partial correctness of the translation of a language called C^{int}, i.e. a WHILE-language with (un-nested) parameterless procedures, to Transputer code (see [24] for details). The recipe was almost clear, [61] provided a clear and transparent view on the behavior of the Transputer and [60] showed that for preservation of partial correctness it seems unavoidable to induce on a functional describing the semantics of the executing machine. A first very naive attempt was [85] which considers the abstract level of [60] but extended by procedures. Now, with the present thesis most of the ingredients seem to be carried together. The appendix adds procedures to the abstract view on the behavior of the Transputer and apart from several details the essences of the actual partial correctness proof on a clean level are given here. However, the actual work is still to be done.

Extensions. We considered the control flow aspects of strictly transformational imperative programs. It would be interesting to study how relative correctness gets along with other paradigms. Some of the ideas and results presented in Chap. 5 might be used for reasoning about process-programming, reactive or other systems that, e.g., must not stop and thus have no "outcomes" except for '∞' which is obviously not adequate in these concerns.

Mechanical Support. As usual there is the desire for proof assistance, and in this particular case it comes in two flavors. Not solely for the reasoning about wrp-semantics but generally for calculations in an equational style it would be advantageous to have a tool at hand which supports ad-hoc reckoning and prototyping. Having found a proof it would also increase credibility if the proof could be rerun or checked automatically ([80] is a nice approach but wrapped in a very

specific context). Furthermore and especially in the environment of the *Verifix* project it is necessary to document proofs, and this in a comprehensible manner. One of the Ulm-group's jobs is to generate proofs resp. to reproduce hand-waved proofs (almost) automatically using the PVS system. As most of the actual proofs have an operational flavor the generated proof protocols are barely plain, not to mention short. A predicate transformer semantics would dramatically increase readability and thus it is advisable to improve on this matter. First steps have been done, e.g. [69], others are in preparation but there is still a lot to do.

Further wrp-Laws. As mentioned before, we think wrp-based reasoning has demonstrated its practicability and elegance for more realistic questions concerning program and translation verification. Therefore, it would be nice to have a more profound pool of wrp-rules available. In Chaps. 6 and 8 we provided the basic rules and also some more detailed ones which are intended for a specific goal, but a rich collection like, e.g., [53] or in [61] would pave the way for even more comprehensible proofs and legitimations of optimizations.

A wrp- resp. Relative-Correctness-Calculus. The wrp-transformers correspond to relative correctness w.r.t. A which is a generalization of partial correctness. The Hoare-calculus – dealing with partial correctness – is known to be correct and relatively complete w.r.t. the common denotational semantics of WHILE-languages. The key to relative completeness, however, is the assumption that expressions cannot evaluate erroneously. If this is not the case then relative correctness cannot be shown but it can be achieved anyhow by guarding the assignment-axiom in the shape of $\{\mathsf{def}(e) \to \phi[e/x]\}x := e\{\phi\}$. It would be very interesting to research on this topic and to extend insights to relative correctness.

General Questions. The high-level language presented in Sect. 7.4 is equipped with a static scoping semantics. It turned out to be non-trivial to define an operational semantics but the denotational semantics is quite clear and elegant. On the other hand, an operational semantics modeling dynamic scoping can easily be specified. What should a denotational semantics for dynamic scoping look like?

We cannot get rid of the impression that showing preservation of total correctness is easier than showing preservation of partial correctness. As mentioned in Sect. 8.3, there are some formal evidences but can the impression indeed be confirmed or are appearances deceiving? As the case may be, but why?

A. Reflections on the Toy-Assembly-Language

The present appendix is intended to justify a seemingly bald statement made in Chap. 7. On p. 95 we claimed that the assembly language presented in Sect. 7.3 is *almost* an abstract view on an existing language, namely the Transputer code [37]. Of course this can hardly be accepted without further explanations and now is the time and this is the place for a justification. The oncoming discussion is barely self-contained because a vast number of notions, definitions and mostly notations have to be introduced, thus it is shifted to this appendix rather to an own chapter or section. Moreover, the mentioned preparations are not fulfilled at all; the needed vocabulary and technical means is purely cited from [61] and we will try to make them plausible using our own vocabulary to the best of our knowledge and in all conscience. In this sense the appendix should be read only with a copy of [61] nearby. For a supporting literature we also recommend [21] which presents a shorter (w.r.t. [61]) but even more detailed (w.r.t. the present) exposition and from which we borrow a lot of phrases. But before going into details let us set the stage and start with a motivation.

A.1 A Prefacing View

In his doctoral dissertation [61] Markus Müller-Olm presented a correctness proof for a translation of a real-time WHILE-language to the Transputer code, to be precise he proved the translation correct in the sense of preservation of total correctness. This is an interesting result, yes, but more exciting – and this is the actual story told there – is the way this proof is performed. Code generator correctness proofs may easily get monolithic as typically a specific compilation scheme for concrete source- and target-languages is at hand which has to be verified. Obtained results may be of particular interest but not the actual proofs as they typically are not reusable in no way; perhaps apart from some used proof-techniques in isolation. Whenever the source- or target-language is modified, even if very slightly, the whole proof has to be rerun. This is obvious and in some sense unavoidable but the effects become really unattractive if one realizes that the entire proof is futile just because some odds and ends are added or omitted.

The remedy proposed in [61] is modularity and abstraction. Starting from real bit-code, the semantics of which is taken directly from the Transputer manual [37], running on the real Transputer resp. on its diverse components (like registers, flags, memory and pointers), via various intermediate steps, an abstract level of machine-description, an assembly language, is derived in which most of

the components of the Transputer have vanished but in which it is still possible to reason about the behavior of instructions and programs in a given context. In so doing, increasingly more conceptual and general but still consistent, i.e. semantics preserving, views on the behavior of the Transputer are evolved and each abstraction step allows to tackle one specific phenomenon in isolation and thus more comprehensible. The technical means that is used is, roughly speaking, the space of predicate transformers. An imperative meta-language in the shape of the refinement calculus is defined which is interpreted by Dijkstra's wp-calculus and which – as we know – corresponds to total correctness. Furthermore, a variant of well-known data refinement techniques [4, 22, 57] is drawn up that in some sense complies with the media presented in Sect. 5.1 but which is defined in terms resp. is part of the meta-language and thus more handy in [61]. This setup allows to distinct between resp. to relate different state spaces and to reason about implementations. To be slightly more precise, terms in the shape of Theorem 5.2.1 serve for a definition of data-refinement of commands and this notion is used to show that certain instructions have "the same" meaning but seen from different and in the end very abstract and thus more manageable perspectives.

Finally, the idea is – among others – that one uses this more abstract view in the actual translation correctness proof as it provides a facile and more intuitive but still sufficiently detailed imagination of the behavior of the Transputer and by this eases the proof in large amounts. Furthermore, if, for instance, components of the source language are modified or even if the target language of the translation is changed, parts of the previously done work might be reused. In the first case the task is to find instruction sequences of the abstract view that implement the commands of the source language (here the abstract view is recycled); in the second case one tries to "concretize" from the abstract view towards the new target language (and here it is the actual translation).

Let us return to our claim. The assembly language presented in Sect. 7.3 is of course a toy-language but it can – in some sense – be kept for an even more abstract view on the Transputer. Having a look at some concrete results of [61] the suspicion augments that it should be possible to replace our commands by specific Transputer instructions or sequences thereof such that the latter mentioned implement them. This might be plausible for the commands in isolation but if one tries to prove this, the problem pops up that the underlying operational semantics given in Sect. 7.3, in particular the procedure concept, seems to be really wondrous and artificial.

The remainder of this appendix is devoted to a more thorough discussion and justification why our assembly language is slightly more concrete and realistic than it seems to be at first glance. Though we will not prove that the assembly language *is* an abstract view – in fact, it is not yet – we nevertheless want to legitimate the claim that it is *almost* an abstract view and by this that the language considered here can be kept for "a stepping stone on the way down to actual binary machine code".[1]

[1] Thanks to Markus Müller-Olm for this concise and vivid phrase.

A.2 Extensions

As mentioned in the beginning we cannot go into the very details here as we would have to cite more than half of [61]. Instead we revisit the mentioned book directly in the sense that we have a closer look at the proceeding presented there using its own language (for a better guidance we also use the same headings and most of the numbering). For the sake of comprehension – which we barely can expect due to lack of vocabulary – we will nevertheless try to explain and motivate most of the performed steps in our words whenever this is possible and necessary. Furthermore, we extend the work of [61] such that the procedure concept of our assembly language becomes a part of the abstractions, too. Therefore, some more Transputer instructions have to be added, ones that are not needed in [61], and one more abstraction step has to be performed. The resulting final view on the Transputer which includes our procedure concept hopes to illustrate and to increase the appreciation of our vision that the assembly language of Sect. 7.3 is rather natural and not that seemingly artificial and just convenient.

A.2.1 Transputer base model

Appendix F of the Transputer manual [37] describes the components and the instructions of the Transputer semi-formally. The Transputer is essentially a state-machine working on a state consisting of three registers `A`, `B` and `C` (which are typically used as a small stack by most of the instructions), an operand register `Oreg` (providing word-size operands for the instructions), a workspace-pointer `Wptr` (relatively to which instructions typically address), an error-flag `EFlag`, an instruction pointer `IP`, and an addressable memory `Mem`. The behavior of the instructions is described with the aid of Z-like schemata, i.e. a relational pre- and post-style description of the effect of the instructions on the state space. In [61], on the other hand, the effect of instructions is captured by refinement axioms which express changes of states in terms of (multiple) assignments, i.e. assignment predicate transformers. The instruction `ldc(1)`, for instance, which loads the constant value `1` to register `A` and moves `A`'s and `B`'s contents to register `B` and `C` respectively, can be represented by the multiple assignment[2]

$$E_0(\texttt{ldc(1)}) \geq \texttt{A,B,C} := \texttt{1,A,B} \ ,$$

where the predicate transformer $E_0(\texttt{ldc(1)})$ suggests that this is the *definition* of `ldc(1)`'s *effect* and which is precisely what the Transputer manual describes relationally. As the Transputer provides only four bits for coding so-called *direct functions* there exists a particular direct function, `opr`, which executes some further so-called *operations* whose opcode is assumed to be contained in the operand register. To be precise, the effect of the `opr` instruction is given by[3]

[2] In [61] the function-name `wp` is generally omitted. In this sense, $\texttt{A,B,C} := \texttt{1,A,B}$ conforms with $\texttt{wp.(A,B,C} := \texttt{1,A,B)}$. Note furthermore that not equalities are given but inequalities which has the particular benefit that they correspond to safe approximations or under-specifications, i.e. they define what happens at least and say nothing about effects that are not documented in [37].

[3] For a predicate ϕ the *assertion* $\{\phi\}$ is defined by $\{\phi\}.\psi = \phi \wedge \psi$ and represents a process that terminates immediately without an effect if ϕ holds and behaves chaotically otherwise.

$$E_0(\texttt{opr}) \geq \{CurOpr(op)\} \; ; \; E_0(op) \; ; \; \texttt{Oreg} := 0 \;\; ,$$

where

$$CurOpr(op) \iff \texttt{Oreg} = OpCode(op)$$

and $E_0(op)$ is the effect of operation op.

The watchful reader will perhaps miss the typical increment of the instruction pointer IP and the cause for its absence is the following. In [61], the complete behavior of the running phase is modeled by a *Run*-process, i.e. a predicate transformer, which cyclically executes a *Step*-process ($Run = Step \; ; \; Run$) which itself – very roughly – fetches the instruction the instruction pointer IP points at, executes it and increases the instruction pointer ($Step \geq \{CurFct(instr)\} \; ; \; Fetch \; ; \; E_0(instr)$). The increment of IP is treated in the fetch-phase (*Fetch*) and can resp. must be omitted in the definition of an effect $E_0(instr)$.

Besides the instructions considered in [61] the following instructions are needed in order to integrate our rather abstract view on stacking return-addresses. There are the direct functions

$$\begin{aligned} E_0(\texttt{ldnl}) &\geq \{Index(\texttt{A},\texttt{Oreg}) \in Addr\} \; ; \\ &\quad \texttt{A},\texttt{Oreg} := \texttt{Mem}(Index(\texttt{A},\texttt{Oreg})),0 \\ E_0(\texttt{stnl}) &\geq \{Index(\texttt{A},\texttt{Oreg}) \in Addr\} \; ; \\ &\quad \texttt{Mem}(Index(\texttt{A},\texttt{Oreg})),\texttt{A},\texttt{B},\texttt{C},\texttt{Oreg} := \texttt{A},\texttt{C},?,?,0 \\ E_0(\texttt{ldnlp}) &\geq A,\texttt{Oreg} := Index(\texttt{A},\texttt{Oreg}),0 \;\; , \end{aligned}$$

and there is the operation

$$E_0(\texttt{gcall}) \geq \texttt{A},\texttt{IP} := \texttt{IP},\texttt{A} \;\; .$$

Here, $\{Index(\texttt{A},\texttt{Oreg}) \in Addr\}$ ensures that the inequality is trivial if the referenced memory address $Index(\texttt{A},\texttt{Oreg})$[4] is invalid. Note that the *general call* gcall only exchanges the contents of A and IP. Thus, for an implementation of a procedure call, A is assumed to contain the jump-address and afterwards the return address is again contained in A; remember that the fetch-phase increases the instruction pointer. The non-deterministic assignment C :=? expresses that the contents of register C is left unspecified after execution, it will contain a value but in [37] the designers did not say which.

This base-model of the behavior, given by means of predicate transformers, is already a safe abstraction but it is a rather low level of description. Though it is possible to relate a semantics of a source language to the semantics given by E_0 directly, this would be a very clumsy and error-prone task. Instead, more abstract views on the behavior of the Transputer are derived which promise to ease the proof and to be more manageable and intuitive in general.

A.2.2 Symbolic representation of control point

In a first step the progress of program execution is represented symbolically. The idea is that a Transputer program m can be partitioned into two sequences u

[4] Formally, $Index(x,y) = x + bpw * y$ is the word address y words past the base address x.

and v such that $u \cdot v = m$ and the pair $\langle u, v \rangle$ can be kept for the program m in a stage where the first instruction of v is to be executed next. This abstraction lets us forget about the instruction pointer IP and we already know this view on programs from our assembly language, Sect. 7.3. Technically this proceeding is expressed by means of data representation Galois connections, cf. Def. 5.1.1, and refinement inequalities in the shape of Theorem 5.2.1. To be more precise, a family of processes $I_1(u, v)$ is defined which describes the behavior of program $u \cdot v$ if started with the first instruction of v, i.e.

$$I_1(u, v) \stackrel{\text{def}}{=} \mathtt{IP}^+ \,;\, [\Lambda(u, v)] \,;\, \mathit{Run} \,;\, \mathtt{IP}^- \ .$$

Here,[5]

$$\Lambda(u, v) = \mathit{Loadad}(u \cdot v) \wedge \mathit{IPAfter}(u)$$

describes – by means of predicates that are omitted here – that $u \cdot v$ is the program in question ($u \cdot v$ is part of the program storage which is given by a start address s_p and a length l_p) and that *Run* is started in a state where IP points to the first instruction of v (i.e. the first instruction "after" u). Predicate $\Lambda(u, v)$ is called a *coupling-invariant* because it couples two state spaces, the one that knows about the instruction pointer IP but nothing about the abstract view in terms of $\langle u, v \rangle$ and the reverse one. The block $\mathtt{IP}^+ \ldots \mathtt{IP}^-$ hides the instruction pointer that will be no longer needed in the more abstract view. For each instruction *instr* of the Transputer and each corresponding effect process $E_0(\mathit{instr})$ abstractions are defined which ensure that changes in the loaded program or a change of the position of execution lead to chaotic, unpredictable behavior:

$$E_1(\mathit{instr}) \stackrel{\text{def}}{=} \bigwedge_{u,v} \mathtt{IP}^+ \,;\, [\Lambda(u, v)] \,;\, E_0(\mathit{instr}) \,;\, \{\Lambda(u, v)\} \,;\, \mathtt{IP}^- \ .$$

In our vocabulary $\mathtt{IP}^+ \,;\, [\Lambda(u, v)]$ corresponds to the upper adjoint and $\{\Lambda(u, v)\} \,;\, \mathtt{IP}^-$ to the lower adjoint of a data representation Galois connection, cf. Def. 5.1.1, and by Theorem 5.2.1 the above abstraction defines $E_0(\mathit{instr})$ to be a correct implementation of $E_1(\mathit{instr})$ in the sense of preservation of total correctness. In other words, $E_1(\mathit{instr})$ is a correct abstraction of $E_0(\mathit{instr})$ in the same sense. But note that the control flow has not been modified but only its representation. Remember furthermore that $\Lambda(u, v)$ is parameterized with u and v and couples both state spaces. Thus we assume that *instr* is the first instruction of v and assert that afterwards the modeled instruction pointer is still pointing at the same position as its increment is part of the fetch-phase which itself is part of the *Run*-process. Taking the greatest lower bound over all u and v guarantees this abstraction to be valid for all instances of u and v, to hold in all possible contexts.

Then a general instruction theorem (Theorem 10.1.1 of [61]) can be shown,

$$I_1(u, \mathit{instr}(n) \cdot v) \geq \mathtt{Oreg} := \mathtt{Oreg}\ \mathtt{bitor}\ n \,;\, E_1(\mathit{instr}) \,;\, I_1(u \cdot \mathit{instr}(n), v) \ ,$$

where $\mathit{instr}(n)$ denotes the code sequence consisting of the single instruction *instr* with a four-bit operand n, $0 \leq n \leq 16$. It formally reflects the intuition that an

[5] For a predicate ϕ the *assumption* $[\phi]$ is defined by $[\phi].\psi = \phi \longrightarrow \psi$ and represents a process that terminates immediately without an effect if ϕ holds and leads to miraculous success otherwise.

instruction *instr* with operand n started in a context u and v behaves as follows: Firstly n is bitwise or-ed (n is loaded into the least four bits of the operand register `Oreg`), secondly the effect of *instr* is executed itself, and finally control is transfered to the subsequent instruction which is the first of v in this case. The latter is again beautifully expressed by moving $instr(n)$ from the right-hand-side to the end of the left-hand-side (the reader should be acquainted with this).

For the direct functions introduced before the following safe abstractions can be shown.[6]

$$\begin{aligned} E_1(\texttt{ldnl}) &\geq \{Index(\texttt{A},\texttt{Oreg}) \in Addr\}\ ; \\ &\quad \texttt{A},\texttt{Oreg} := \texttt{Mem}(Index(\texttt{A},\texttt{Oreg})),0 \\ E_1(\texttt{stnl}) &\geq \{AdmAddr(Index(\texttt{A},\texttt{Oreg}))\}\ ; \\ &\quad \texttt{Mem}(Index(\texttt{A},\texttt{Oreg})),\texttt{A},\texttt{B},\texttt{C},\texttt{Oreg} := \texttt{B},\texttt{C},?,?,0 \\ E_1(\texttt{ldnlp}) &\geq A,\texttt{Oreg} := Index(\texttt{A},\texttt{Oreg}),0 \end{aligned}$$

As the control flow is really changed by the `gcall`-operation it has a (sensible) meaning only in a surrounding context which allows to express jump-destinations. Similar to Theorem 10.1.10 of [61] where (un)conditional jumps are considered and also similar to our semantics-definition in Sect. 7.3 the following holds.

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = c \cdot d$ *and* $j = s_p + |c|$, *where* $\#\texttt{gcall} = OpCode(\texttt{gcall})$. *Then*

$$\begin{aligned} &[\texttt{Oreg} = 0]\ ;\ [\texttt{A} = j]\ ;\ I_1(a, \texttt{opr}(\#\texttt{gcall}) \cdot b) \\ &\quad \geq \texttt{A},\texttt{Oreg} := s_p + |a| + 1, 0\ ;\ I_1(c,d)\ . \end{aligned} \tag{A.1}$$

This theorem conforms with the intuition and beautifully expresses that control is transfered to the first instruction of d, the position of which is determined by the jump-address j and that the return address is saved in register `A`.

A.2.3 Large operands

In a second abstraction step one gets rid of the operand register `Oreg`. Its purpose is to provide word-size operands for the instructions which typically are filled in portions of four bits by sequences of so-called `pfix` and `nfix` instructions. As yet, `Oreg` is treated like any other register so the next abstraction step provides an understanding of leading `pfix` and `nfix` chains together with another leading instruction serving for a multi-byte instruction.

It turns out that one is interested in starting code sequences with a cleared operand register, e.g. (A.1), as this allows to handle jumps adequately. This suggests a data refinement which is, for processes P, defined by

$$G_2(P) \stackrel{\text{def}}{=} \texttt{Oreg}^+\ ;\ [\texttt{Oreg} = 0]\ ;\ P\ ;\ \{\texttt{Oreg} = 0\}\ ;\ \texttt{Oreg}^-\ .$$

Again, the operand register is hidden as it will be of no further interest in the sequel. The new, i.e. more abstract, run-phase of the Transputer is defined by

[6] An approximation for instructions that assign to the memory must ensure that the program storage is not corrupted. This expresses in the predicate *AdmAddr* which holds for a word w iff w belongs to the accessible addresses *Addr* but does not point into the program storage.

$$I_2(u,v) \stackrel{\text{def}}{=} G_2(I_1(u,v)) \ ,$$

hence exactly by the old one, just forget about `Oreg` if the input starts with a cleared and ends with a cleared operand register. As an entire prefixed instruction $instr(w)$[7] loads the operand register with w and behaves like $instr$ afterwards it is proximate to define the effect of instructions on this level by[8]

$$E_2(instr,w) \stackrel{\text{def}}{=} G_2(\texttt{Oreg} := w \ ; E_1(instr)) \ ,$$

because by this definition the general instruction theorem reduces to

$$I_2(u, instr(w) \cdot v) \geq E_2(instr,w) \ ; I_2(u \cdot instr(w), v) \ .$$

Applying the abstraction E_2 to our newly integrated direct functions yields

$$\begin{aligned} E_2(\texttt{ldnl},w) &\geq \{Index(\texttt{A},w) \in Addr\} \ ; \\ &\quad\ \texttt{A} := \texttt{Mem}(Index(\texttt{A},w)) \\ E_2(\texttt{stnl},w) &\geq \{AdmAddr(Index(\texttt{A},w)\} \ ; \\ &\quad\ \texttt{Mem}(Index(\texttt{A},w)), \texttt{A}, \texttt{B}, \texttt{C} := \texttt{B}, \texttt{C}, ?, ? \\ E_2(\texttt{ldnlp},w) &\geq A := Index(\texttt{A},w) \ , \end{aligned}$$

and the theorem about `gcall`, (A.1), transfers to the present view as follows (cf. Theorem 10.2.7 in [61]).

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = c \cdot d$ *and* $j = s_p + |c|$, *where* $\#\texttt{gcall} = OpCode(\texttt{gcall})$. *Then*

$$\begin{aligned} &[\texttt{A} = j] \ ; I_2(a, \texttt{opr}(\#\texttt{gcall}) \cdot b) \\ &\quad \geq \texttt{A} := s_p + |a| + 1 \ ; I_2(c,d) \ . \end{aligned} \tag{A.2}$$

As expected, nothing changed what the control flow concerns, we just have a more facile but still consistent view on the behavior.

A.2.4 Workspace

The third abstraction step presented in [61] replaces the memory `Mem` by a workspace `Wsp` of fixed length, l_w, and with a fixed memory-cell, s_w, which is intended to represent the position of the workspace pointer. This workspace (which can be taken for an array) makes reasoning about the memory more convenient as only a distinguished part of it is of interest. Essential for the abstraction to be performed is an overall assumption resp. requirement that all workspace locations have admissible addresses which implies that workspace and program storage are disjoint by (here omitted) definition. This global requirement is assumed to be established by a loader and the instructions have to leave this requirement invariant. A further observation makes the abstraction sensible at all. None of the

[7] Intuitively, $instr(w)$ represents a sequence of instructions that applies the direct function $instr$ to the word operand w, more precisely it is a sequence of pfix and nfix instructions suitably coded by w followed by the instruction $instr$ applied to a four-bit remainder of w. For further explanations and results see [61].

[8] Some timing constraints are omitted.

instructions considered here modifies the workspace pointer `Wptr` so it is reasonable to assume the position of `Wptr` being fixed. All in all, we suggest the following predicate for a coupling invariant[9]

$$
\begin{aligned}
&\mathit{WspInMem} \iff \\
&\quad \mathtt{Wptr} = s_w \wedge \\
&\quad \forall i:\ 1 \le i \le l_w:\ \mathtt{Wsp}[i] = \mathtt{Mem}(\mathit{Index}(s_w, i)) \wedge \\
&\quad \forall i:\ 1 \le i \le l_s:\ \mathtt{Sys}[i] = \mathtt{Mem}(\mathit{Index}(s_w, i)) \wedge \\
&\quad \forall i:\ 1 \le i \le l_v:\ \mathtt{Vars}[i] = \mathtt{Mem}(\mathit{Index}(s_w, l_s + i)) \wedge \\
&\quad \forall i:\ 1 \le i \le l_r:\ \mathtt{Raddr}[i] = \mathtt{Mem}(\mathit{Index}(s_w, l_s + l_v + i))
\end{aligned}
$$

In [61] the workspace `Wsp` is introduced in order to provide a clear and transparent view on the memory such that it can easily be replaced by a list of symbolic variables in a next abstraction step. As we do also head for a comprehensible view on the implementation of procedures we add some more components. Like in [61], the *WspInMem*-predicate defined above still specifies a distinguished memory-area called `Wsp` and the workspace pointer is also assumed to be fixed at position s_w. In our view, however, we furthermore assume the workspace `Wsp` to be partitioned into three zones: `Sys` may store some system variables that might be necessary later, `Vars` is supposed to store variables and will thus play the role of the entire `Wsp` in [61], and `Raddr` will be used to store return addresses. The actual abstraction is defined by[10]

$$
\begin{aligned}
&G_3(P) \stackrel{\text{def}}{=} \\
&\quad \mathtt{Wptr}, \mathtt{Mem}^{+}\ ;\ [\mathit{WspInMem}]\ ;\ \mathtt{Wsp}, \mathtt{Sys}, \mathtt{Vars}, \mathtt{Raddr}^{-}\ ;\ P\ ; \\
&\quad \mathtt{Wsp}, \mathtt{Sys}, \mathtt{Vars}, \mathtt{Raddr}^{\oplus}\ ;\ \{\mathit{WspInMem}\}\ ;\ \mathtt{Wptr}, \mathtt{Mem}^{-}\ ,
\end{aligned}
$$

and consequently the behavior of partitioned code sequences and prefixed instructions is given by

$$
\begin{aligned}
I_3(u, v) &\stackrel{\text{def}}{=} G_3(I_2(u, v)) \\
E_3(\mathit{instr}, w) &\stackrel{\text{def}}{=} G_3(E_2(\mathit{instr}, w))\ .
\end{aligned}
$$

The results concerning the instructions and operations presented in [61] remain valid as we performed the same abstraction and only added some more variables to the present view on the Transputer. And so does the general instruction theorem (Theorem 10.3.1), it reads

$$
I_3(u, \mathit{instr}(w) \cdot v) \ge E_3(\mathit{instr}, w)\ ;\ I_3(u \cdot \mathit{instr}(w), v)\ .
$$

The newly added instructions on the other side look quite different now as they do not address w.r.t. the workspace pointer but w.r.t. register `A`.

If $\mathit{Index}(\mathtt{A}, w)$ is an address then $\mathtt{Wsp}[\frac{\mathtt{A}-s_w}{bpw} + w] = \mathtt{Mem}(\mathit{Index}(\mathtt{A}, w))$ and if furthermore $\frac{\mathtt{A}-s_w}{bpw} + w$ lies in the workspace, i.e. if $1 \le \frac{\mathtt{A}-s_w}{bpw} + w \le l_w$, then

[9] In [61] the latter three conjuncts are non-existent.

[10] Here, $var^{\oplus}$ denotes angelic addition of variable *var* to the state space, var^{+} is the demonic addition. Both have an operational interpretation and also a theoretical background, see [61].

$$\begin{aligned} E_3(\texttt{ldnl}, w) &\geq \texttt{A} := \texttt{Wsp}[\frac{\texttt{A} - s_w}{bpw} + w] \\ E_3(\texttt{stnl}, w) &\geq \texttt{Wsp}[\frac{\texttt{A} - s_w}{bpw} + w], \texttt{A}, \texttt{B}, \texttt{C} := \texttt{B}, \texttt{C}, ?, ? \\ E_3(\texttt{ldnlp}, w) &\geq A := \mathit{Index}(\texttt{A}, w) \ . \end{aligned}$$

A – sensible – `gcall`-theorem for this view can also be given but the situation is quite more complicated now. A look at (A.2) reveals that the return address is saved in register `A` but of course it may immediately be overwritten and thus it may get lost before a subsequent return is reached. The idea is to store the return address in the workspace, in the `Raddr`-zone to be precise. As there is no single instruction for a proceeding like this we have to implement it manually. Of course we have to implement a stacking-mechanism as the stored return addresses must be accessible even if a new procedure is called. So suppose that one of the system variable cells, say $\texttt{Sys}[rap]$ with $1 \leq rap \leq l_s$, is used as a *return address pointer*, a pointer that points to the top of the return address stack that is to be implemented in the `Raddr`-zone now. Then consider the sequence below that uses only instructions that are available:

$$\mathit{entrycode} \stackrel{\text{def}}{=} \texttt{ldl}(rap) \cdot \texttt{ldnlp}(1) \cdot \texttt{stl}(rap) \cdot \texttt{ldl}(rap) \cdot \texttt{stnl}(0) \ .$$

Roughly speaking, *entrycode* has the following effect. Firstly the content of the return address pointer (this is the address of the top-of-the-stack) is loaded into register `A` and `A`'s content is moved to `B`. Afterwards `A`'s content, i.e. the content of the return address pointer, is increased by one and saved to the return address pointer again. Additionally `B`'s content is moved back to `A`. Then the modified content of the return address pointer, i.e. the increased one, is loaded into `A` again, `A`'s content is thereby moved to `B`, and finally `B`'s content is saved to the workspace cell specified by `A`. Thus, if we assume that `A` initially keeps the current return address then *entrycode* stores it in a new cell of the `Raddr`-zone (we omitted to check for overflows here). Indeed this can be proved algebraically. The aspired `gcall`-theorem reads as follows.

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = c \cdot \mathit{entrycode} \cdot d$, $j = s_p + |c|$ *and* $r = s_p + |a| + 1$ *where* $\#\texttt{gcall} = \mathit{OpCode}(\texttt{gcall})$. *Then*

$$\begin{aligned} &[\texttt{A} = j] \ ; I_3(a, \texttt{opr}(\#\texttt{gcall}) \cdot b) \\ &\quad \geq \texttt{Sys}[rap], \texttt{Wsp}[\frac{\texttt{Sys}[rap] - s_w}{bpw} + 1] := \mathit{Index}(\texttt{Sys}[rap], 1), r \ ; \\ &\quad \texttt{A}, \texttt{B}, \texttt{C} := \texttt{B}, ?, ? \ ; \\ &\quad I_3(c \cdot \mathit{entrycode}, d) \end{aligned} \tag{A.3}$$

This looks nice but still quite complicated. However, if we want to resolve a return address that has been stored on the stack before we need a suitable sequence of instructions which somehow reverses the effect of *entrycode*. A promising candidate is[11]

[11] Essentially, the idea of *entrycode* and *exitcode* is taken from [24] which considers the authentic translation from C^{int} to the Transputer, see also p. 173.

$$exitcode \stackrel{\text{def}}{=} \texttt{ldl}(rap) \cdot \texttt{ldnlp}(-1) \cdot \texttt{stl}(rap) \cdot \texttt{ldl}(rap) \cdot \texttt{ldnl}(1) \ .$$

Assuming that *exitcode* indeed resolves the return address and leaves it in register `A` a procedure-return can be implemented by yet another `gcall`. This motivates to define

$$\texttt{return} \stackrel{\text{def}}{=} exitcode \cdot \texttt{gcall} \ ,$$

and, in fact, our intuition was right. The following observation can be shown which holds slightly more general than presented here. We decided to state this version in order to demonstrate how it fits to (A.3).

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = e \cdot \texttt{return} \cdot f$, $n = s_p + |e \cdot \texttt{return}| + 1$ *and* $r = s_p + |a| + 1$ *where* $\#\texttt{gcall} = OpCode(\texttt{gcall})$. *Then*

$$\begin{aligned} &[\texttt{Wsp}[\frac{\texttt{Sys}[rap] - s_w}{bpw}] = r] \ ; I_3(e, \texttt{return} \cdot f) \\ &\quad \geq \texttt{Sys}[rap], \texttt{A}, \texttt{B}, \texttt{C} := Index(\texttt{Sys}[rap], -1), n, \texttt{A}, \texttt{B} \ ; \\ &\quad\ I_3(a \cdot \texttt{opr}(\#\texttt{gcall}), b) \end{aligned} \tag{A.4}$$

Thus, the return address stored in $\texttt{Wsp}[\frac{\texttt{Sys}[rap] - s_w}{bpw}]$ by a `gcall` to an *entrycode* is resolved if subsequently a `return` is executed; note that in (A.3) the pointer is simultaneously increased such that this interpretation conforms with the calculation. However, this looks still quite sophisticated and hard to handle in a translation-correctness proof. In a while, after one more abstraction step, the situation gets clearer and the reader hopefully appreciates our proceeding.

A.2.5 Symbolic addressing

In the fourth abstraction step presented in [61] the workspace is completely replaced by a list of symbolic variables. The variables which are introduced by programs of the considered source language in [61] and their representations in the workspace are related by variable dictionaries δ. Omitting some further definitions the coupling invariant reads

$$InWsp^{\delta} \stackrel{\text{def}}{=} \bigwedge_{x \in \text{dom}(\delta)} (x, \texttt{Wsp}[adr_x]) \in R_x \ ,$$

and roughly verbalized it has the following meaning. For variables x with value domain T_x a so-called representation relation $R_x \subseteq T_x \times \texttt{Word}$ is defined such that $(t, w) \in R_x$ means that the word w stored in a workspace cell adr_x allocated for x is a proper representation for the value t stored by x.

This abstraction does not directly transfer to our view as we dot not want to replace the entire workspace, we already implemented a return-address stack which would be destroyed. However, we provided the workspace with a specific area, `Vars`, which is intended to store variables and this fourth abstraction step is – in principle – applicable in our scenario, too, by taking

$$InVars^{\delta} \stackrel{\text{def}}{=} \bigwedge_{x \in \text{dom}(\delta)} (x, \texttt{Vars}[adr_x]) \in R_x$$

for the coupling invariant and by carefully and attentively modifying definitions and argumentations here and there wherever it is necessary. In particular some rules concerning certain direct functions, e.g. `ldl` and `stl`, need further premises.

We skip the remainder of this abstraction step as we are interested in the procedure mechanism mainly. Let us assume that we will be able to integrate variables into our setting – [61] will hopefully guide us through this task – and let us perform yet another abstraction, one that is not made in [61] as it is not needed there (the source language considered there knows nothing about procedures).

A.2.6 Abstract return addresses

The abstractions made so far let us reason about Transputer programs in a very intuitive way. The control flow, initially given by means of `IP`-movements, is now represented by partitioning the program in question differently, and most of the data flow is described in terms of dictionaries. Less intuitive is the stacking-mechanism for return-addresses, in particular there is a mismatch in the sense that progression of execution is modeled abstractly but the return addresses are still given absolutely. Seen from this perspective, it would be more instinctive to store the partitioned code sequence to continue with after a return instead of the absolute address of this continuation point (roughly speaking, one gets rid of the calculation which determines this point). Let us try to model this more abstract view purely algebraically as done for the previous steps.

Assume given a variable `RAS` which serves for an abstract *return address stack* (it will play the role of a in the operational semantics given in Sect. 7.3) each entry of which is a splitted instruction sequence. For every Transputer program m we reasonably expect the following to hold for any point of the execution time. The size of the absolutely given return address stack, i.e. $\frac{\mathtt{Sys}[rap]-s_w}{bpw} - l_s - l_v$, equals the size of the abstract return address stack, i.e. $|\mathtt{RAS}|$. Furthermore, each entry of `RAS`, say $\mathtt{RAS}.i$, is a partition of m which codes the position of the instruction pointer which itself is contained in $\mathtt{Raddr}[i]$. These expectations are more technically expressed in the following predicate.

$$
\begin{aligned}
&\mathit{Stacked}(m) \iff \\
&\quad |\mathtt{RAS}| = \frac{\mathtt{Sys}[rap] - s_w}{bpw} - l_s - l_v \ \wedge \\
&\quad \forall i : \ 1 \le i \le |\mathtt{RAS}| : \\
&\qquad (\forall x, y : \ x \cdot y = m \wedge \mathtt{Raddr}[i] = |x| : \ \mathtt{RAS}.i = \langle x, y \rangle) \ .
\end{aligned}
$$

Based on this coupling invariant which relates both views, the one that knows about `Raddr` but nothing about `RAS` and the one that knows about `RAS` but nothing about `Raddr`, the following data abstraction is defined.

$$
\begin{aligned}
G_5^m(P) \quad \stackrel{\text{def}}{=} \quad & \mathtt{Raddr}^+ \ ; [\mathit{Stacked}(m)] \ ; \mathtt{RAS}^- \ ; P \ ; \\
& \mathtt{RAS}^{\oplus} \ ; \{\mathit{Stacked}(m)\} \ ; \mathtt{Raddr}^-
\end{aligned}
$$

The yet more abstract run-phase is given by

$$I_5(u,v) \stackrel{\text{def}}{=} G_5^{u\cdot v}(I_4(u,v))$$

and similarly is the behavior of functions and operations which are omitted here as we are interested in more abstract versions of the `gcall` theorem (A.3) and the return theorem (A.4) mainly.

Note that we canceled the `Raddr`-zone from our view, the remaining registers are still present. As before register `A` is assumed to contain jump addresses and appropriate instructions have to ensure this assumption in a later stage. The `gcall`-theorem in the present setting reads as follows.

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = c \cdot \mathit{entrycode} \cdot d$ *and* $j = s_p + |c|$ *where* $\#\texttt{gcall} = \mathit{OpCode}(\texttt{gcall})$. *Then*

$$\begin{aligned}&[\texttt{A} = j]\ ;\ I_5(a, \texttt{opr}(\#\texttt{gcall}) \cdot b) \\ &\quad \geq \texttt{RAS} := \texttt{RAS} \cdot \langle a \cdot \texttt{opr}(\#\texttt{gcall}), b\rangle\ ; \\ &\qquad \texttt{A},\texttt{B},\texttt{C} := \texttt{B}, ?, ?\ ; \\ &\qquad I_5(c \cdot \mathit{entrycode}, d)\end{aligned} \tag{A.5}$$

The corresponding (and again demonstratively instantiated) `return`-theorem which illustrates how return addresses are resolved is the below one.

Suppose $a \cdot \texttt{opr}(\#\texttt{gcall}) \cdot b = e \cdot \texttt{return} \cdot f$ *and* $n = s_p + |e \cdot \texttt{return}| + 1$ *where* $\#\texttt{gcall} = \mathit{OpCode}(\texttt{gcall})$. *Then*

$$\begin{aligned}&[\texttt{RAS} = c \cdot \langle a \cdot \texttt{opr}(\#\texttt{gcall}), b\rangle]\ ;\ I_5(e, \texttt{return} \cdot f) \\ &\quad \geq \texttt{RAS},\texttt{A},\texttt{B},\texttt{C} := c, n, \texttt{A}, \texttt{B}\ ; \\ &\qquad I_5(a \cdot \texttt{opr}(\#\texttt{gcall}), b)\end{aligned} \tag{A.6}$$

In comparison with (A.3) resp. (A.4) the intuitive comprehension has dramatically increased and we hope the reader feels the same. A `gcall` to an *entrycode* means to stack the sequence with the modeled `IP` pointing to the successive instruction. If subsequently a `return` is reached this topmost entry is popped off the stack and is taken for the sequence to continue with, the tail of the stack remains untouched. Altogether, this final[12] abstraction step was truly worth while. The next subsection is concerned with the question how we can benefit from this abstract but consistent view on the behavior of the Transputer in respect of our assembly language considered in Sect. 7.3.

A.3 The Moral of the Tale

Let us continue our idealized and superficial discussion concerning the closeness to reality. We will become more vague in the sequel and will only present some ideas because there is still a great distance between the abstract view presented before and our assembly language. Furthermore this appendix is meant to serve for a justification that our language is *almost* an abstract view rather than for a proof

[12] In [61] there is one further abstraction step that forgets about registers. Most of the instructions do not have a meaning of their own anymore and for our rough exposition this abstraction gives no further insight.

that it *is* an abstract view on the Transputer. Actually, the view derived so far does not conform with our language at as, for instance, the latter is embodied with labels serving for jump-destinations whereas the former uses addresses. However, the reader is invited to tape up the following position and follow our thoughts concerning a possible further proceeding.

Firstly it should have become clear that modeling the movements of an instruction pointer by partitioning the program in question differently is a sensible and consistent abstraction. It facilitates reasoning about the behavior of programs in large amounts and there should be no doubt about it. In this sense our semantics is reasonable.

Secondly, and particularly very roughly, we claim that apart from the procedure mechanism all ingredients for an instantiation of our "macros" $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$ can be found in [61]. As a part of the correctness proof the translation of the considered (timed) WHILE-language to real Transputer code is inductively defined. Let us briefly visit some of the rules mentioned there though they will be hardly understandable because there is still a lack of vocabulary.

But before doing so we also affirm that it is possible to omit labels in our assembly language and consequently ban labels from our setting at all. If we assume our instructions – which we keep for "macros" – to be already expanded to sequences of real Transputer instructions or if we assume that each of our instructions has a specific length, then we can express the destinations of (conditional) jumps by means of numbers instead of labels; this is possible because `cj` jumps *relatively* to the content of the operand register. To proceed similar for subroutine calls, note that `gcall` jumps to the *absolute* address contained in register `A`, some more modifications have to be performed. The idea is to let the translation dictionary $\boldsymbol{\delta}$ assign procedure names to relative addresses which act as an offset and to implement the `jsr`-command manually in such ways that it computes the absolute address of the actual jump destination, see below. In so doing, the translation would hopefully remain provably correct though probably this proof would not be as clean as presented here because verifying jumps would amount in a counting-exercise. However, let us assume for the moment that our assembly language is not equipped with labels. Furthermore we freely modify the source language considered in [61] in the sense that we keep it for our target language whenever this seems appropriate. We also omit the timing constraints and we devise some further operations our source language could be provided with in order to reason about expressions.

Theorem 12.3.3 of [61] tells us that the code sequence $m_1 \cdot \mathtt{stl}(adr_x)$ is a (correct) translation of the assignment $\mathsf{asg}(x, e)$ if m_1 is a (correct) translation of the expression e for a reasonable variable dictionary. An expression e which, for instance, consists of an addition, say $e = e_1 + e_2$ is (correctly) translated to the sequence $m_2 \cdot \mathtt{stl}(adr) \cdot m_1 \cdot \mathtt{ldl}(adr) \cdot m_0$ if m_1 and m_2 are (correct) translations of the expressions e_1 and e_2 respectively, if m_0 is the (correct) translation of the binary operator '+' and if some further constraints concerning dictionaries hold, cf. Theorem 12.4.4. Finally, by Theorem 12.6.1 the summation sign '+' is (correctly) translated to operation `add` and we are done.

A similar argumentation can be made for conditional jumps. Assume that m_b is the translation of a Boolean expression b (subsection 12.6.3 of [61] deals with this). Then the instruction sequence $m_b \cdot \mathtt{cjp}(n)$[13] firstly evaluates b and overjumps n subsequent instructions on false or transfers control to the directly successive instruction otherwise, this can be seen from Theorem 12.3.10.

These two examples suggest that it should indeed be possible to keep our assembly instructions $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$ for "macros" which can be instantiated by (sequences of) real Transputer instructions. The fact that we considered instructions embodied with entire (Boolean) expressions is only due to convenience as we focused on the implementation of the control flow; the sequences which implement $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$ are flat and thus boring in this sense.

What remains is a justification of our procedure mechanism so let us comment on this. Comparing (A.5) and (A.6) with our predicate transformer laws for the assembly language, Fig. 7.2, shows that storing whole partitioned instruction sequences serving for an abstract representation of concrete return addresses is not that artificial and wondrous as it seems to be at first glance. We also know a Transputer-code sequence which implements our `ret`-instruction, namely `return`. To implement its counterpart, jsr, some preparatory work has to be done and some further assumptions are to be made. Suppose a specific cell of the `Sys`-zone, say $\mathtt{Sys}[start]$ with $1 \leq start \leq l_s$ and $start \neq rap$, is intended to keep the value of the start address s_p of the program storage which can be thought to be communicated by the loader program. Furthermore assume that the compiling-scheme of Sect. 8.2 is modified in the following ways. Firstly, and we already discussed this, asg, cj and `ret` are replaced by their implementations as suggested before. Secondly, if a block introducing procedure p is translated then the translation of p's body receives a preceding *entrycode* and for this translation as well as for the translation of the body of the block itself, the current translation dictionary gets a new binding for p. It assigns to that number determining the first instruction of the corresponding *entrycode*. Finally, a call of procedure p can be translated to

$$\mathsf{jsr}(\boldsymbol{\delta}(p)) \stackrel{\text{def}}{=} \mathtt{ldl}(start) \cdot \mathtt{ldc}(\boldsymbol{\delta}(\mathtt{p})) \cdot \mathtt{opr}(OpCode(\mathtt{add})) \cdot \mathtt{opr}(OpCode(\mathtt{gcall}))$$

because it can be shown that this sequence leaves $s_p + \boldsymbol{\delta}(p)$, i.e. the absolute start address of p's body (including the preceding *entrycode*) at execution time when the program is loaded into the program storage, in register `A` such that, in the very end, the `gcall`-theorem (A.5) reduces to the following.

Suppose $a \cdot \mathsf{jsr}(n) \cdot b = c \cdot entrycode \cdot d$ *where* $n = |c|$. *If* $\mathtt{Sys}[start] = s_p$ *then*

$$\begin{aligned} & I_5(a, \mathsf{jsr}(n) \cdot b) \\ & \quad \geq \mathtt{RAS} := \mathtt{RAS} \cdot \langle a \cdot \mathsf{jsr}(n), b \rangle \ ; \\ & \qquad \mathtt{B}, \mathtt{C} := ?, ? \ ; \\ & \qquad I_5(c \cdot entrycode, d) \end{aligned}$$

[13] The conditional jump of the Transputer is denoted by `cj`. As it could be easily mistaken for cj of our assembly language we let `cjp` play its role.

We hope that this final definition demonstrates that – in principle and apart from the use of labels which were introduced in order to increase readability and comprehension – all of our "macro"-instructions can be flattened to real Transputer-code such that our language can be kept for – more or less – realistic.

A.4 Remarks

Once again we like to stress that this appendix is meant for a pure justification. We do not claim that the assembly language defined in Sect. 7.3 *is* an abstract view on an existing assembly language, e.g. Transputer code, but we do claim that the language considered there makes sense what the control flow aspects concerns and that particularly the ways its semantics is defined, including the procedure mechanism, is not that artificial as it seems. On contrary, as just observed, this view on progress of execution and stacking of return addresses is indeed an abstraction of a real language.

A few remarks remain to be made because the careful reader may object that [61] is concerned with preservation of total correctness which, as the "rule-of-thumb", Sect. 7.1, suggests corresponds to least fixpoints. The question arises how the results of [61] from which we benefit here apply in the case of preservation of partial correctness and variations thereof where greatest fixpoints are adequate. Let us briefly comment on this. The choice of fixpoints is also relevant in [61] and expresses in some axioms capturing the dynamic behavior of Transputer programs, see Sect. 8.4 in [61]. As already mentioned in Sect. A.2 the complete behavior of the running phase of the Transputer is modeled by a *Run*-process which cyclically executes a *Step*-process. This execution-cycle expresses in the axiom $Run = Step\ ;\ Run$. The emerging problem is that this equation fails to determine *Run* uniquely. As PTC is the notion of interest in [61] one should take the least solution. If one decides to do so, it follows that $Run\ ;\ X = Run\ ;\ Y$ for all processes X and Y; this can be shown by simple application of the induction rule. For stylistic reasons [61] desisted from taking the least solution and takes the property $Run\ ;\ X = Run\ ;\ Y$ for all X and Y as a second – and final – axiom instead. Both axioms together play an important role in the remainder, in particular the general instruction theorems rely on them. By leaving unspecified which solution to take, the greatest solution is also permitted but the remaining question is how the greatest solution gets along with the second property. Luckily, one can show $Run\ ;\ X \leq Run\ ;\ Y$ for all X and Y, and hence also an equality, also for the greatest solution, for instance by fixpoint induction. The base case, i.e. $Run\ ;\ X \leq \top\ ;\ Y$ is obvious and for the induction step assume given a predicate transformer f satisfying $Run\ ;\ X \leq f\ ;\ Y$. Then

$$Step\ ;\ f\ ;\ Y \geq Step\ ;\ Run\ ;\ X = Run\ ;\ X$$

such that $Run\ ;\ X \leq \nu_x(Step\ ;\ x)\ ;\ Y$ follows. Admissibility is again obvious by the distribution properties of ';', see (4.1). Thus, even choosing the greatest fixpoint does not violate the axioms and the results concerning pure data abstractions – and in Sect. A.2 we only used those and no translation theorems – remain valid and

applicable even in the PPC case. However, one should be rather careful not to use other, here not mentioned, consequences that rely on a specific choice of fixpoints or equations. Of course, the actual translation-correctness proof presented in [61] does not transfer to the PPC-case but our second proof presented in Sect. 8.2 is intended to provide a general guidance.

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. K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.
4. R.-J. R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness. REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1989.
5. R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
6. R. Berghammer. Programmiersprachen (in German), 1992. Internal report, University of the German forces, Munich.
7. A. Bijlsma. Calculating with procedure calls. *Information Processing Letters*, 46:211–217, 1993.
8. A. Bijlsma, P. A. Matthews, and J. G. Wiltink. A Sharp Proof Rule for procedures in wp Semantics. *Acta Informatica*, 26:409–419, 1989.
9. A. Bijlsma and C. S. Scholten. Point-free substitution. *Science of Computer Programming*, 27:205–214, 1996.
10. G. Birkhoff. *Lattice Theory*, volume 25 of *Amer. Math. Soc. Collog. Publ.* Amer. Math. Soc., 1940.
11. E. Börger and I. Durdanović. Correctness of compiling occam to transputer code. *The Computer Journal*, 39(1), 1996.
12. J. P. Bowen et al. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 50:128–137, June 1993.
13. J. P. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II project final report: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 59, June 1996.
14. J. W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall International, 1980.
15. J. W. de Bakker and D. Scott. A theory of programs, 1969. unpublished notes.
16. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.
17. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
18. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.
19. R. M. Dijkstra. Computation Calculus – Bridging a Formalization Gap. In *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 151–174. Springer-Verlag, 1998.
20. A. Dold, T. Gaul, W. Goerigk, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, V. Vialard, A. Wolf, and W. Zimmermann. Zwischenbericht Verifix. Technical Report Verifix-ZB99, University of Karlsruhe, Karlsruhe, Kiel, Ulm, 1999.
21. M. Fränzle and M. Müller-Olm. Compilation and synthesis for real-time embedded controllers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
22. P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87, 1991. Also in [55].
23. 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, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.
24. W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical report 9713, University of Kiel, Kiel, Germany, 1998.

25. G. Goos and W. Zimmermann. Verification of compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230. Springer-Verlag, 1999.
26. G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, 1978.
27. D. Gries. Monotonicity in Calculational Proofs. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 79–85. Springer-Verlag, 1999.
28. J. D. Guttman, J. D. Ramsdell, and V. Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8:33–110, 1995.
29. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
30. E. C. R. Hehner. Formalization of time and space. *Formal Aspects of Computing*, 10:290–306, 1998.
31. W. H. Hesselink. *Programs, Recursion and unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
32. W. H. Hesselink. Predicate transformers for recursive procedures with local variables. *Formal Aspects of Computing*, 11:616–636, 1999.
33. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
34. C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C. C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer-Verlag, 1991.
35. C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
36. C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.
37. inmos limited. *Transputer Instruction Set – A Compiler Writer's Guide*. Prentice Hall International, first edition, 1988.
38. D. Jacobs. *General Correctness: A Unification of partial and total correctness*. PhD thesis, Computer Science Dept., Cornell University, 1984.
39. D. Jacobs and D. Gries. General correctness: A unification of partial and total correctness. Technical report TR 84-641, Department of Computer Science, Cornell University, Ithaca, New York, 1984.
40. S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.
41. D. Kozen and M.-C. Patron. Certification of Compiler Correctness Using Kleene Algebras with Tests. In *Computational Logic – CL 2000*, volume 1861 of *LNAI*, pages 568–582. Springer-Verlag, 2000.
42. H. Langmaack. Über vollständig operationell adäquate denotationelle Semantik funktionaler Programmiersprachen (in German). Technical report 8901, University of Kiel, Kiel, Germany, 1989.
43. H. Langmaack. Software engineering for certification of systems: specification, implementation, and compiler correctness (in German). *Informationstechnik und Technische Informatik*, 39(3):41–47, 1997.
44. J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, 1982.
45. M. Lenisa. From Set-theoretic Coinduction to Coalgebraic Coinduction: some results, some problems. *Electronic Notes in Theoretical Computer Science*, 19, 1999.
46. K. Lermer and C. J. Fidge. Compilation as refinement. In *Formal Methods Pacific'97*, pages 142–164. Springer-Verlag, 1997.
47. K. Lermer and C. J. Fidge. Subroutine compilation in the refinement calculus, 1999. Submitted to FACS journal.
48. J. Loeckx and K. Sieber. *The Foundations of Program Verification*. John Wiley & Sons and B. G. Teubner, second edition, 1987.
49. Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.
50. O. Mayer. *Syntaxanalyse (in German)*. Bibliographisches Institut, 2nd edition, 1982.
51. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwarz, editor, *Proc. Symp. Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.
52. J. D. Monk and R. Bonnet. *Handbook of Boolean Algebras*, volume 1. North-Holland, Amsterdam, 1989.
53. C. Morgan. *Programming from Specifications*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
54. C. Morgan and A. McIver. Unifying wp and wlp. Technical report 95-36, Department of Computer Science, University of Queensland, Queensland, Australia, 1995.
55. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994.
56. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
57. J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.
58. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

59. S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.
60. M. Müller-Olm. An exercise in compiler verification. Available from the author, 1995.
61. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
62. M. Müller-Olm and A. Wolf. On excusable and inexcusable failures: Towards an adequate notion of translation correctness. In J. Wing, J. Woodcock, and J. Davies, editors, *FM '99 – Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1107–1127. Springer-Verlag, 1999.
63. M. Müller-Olm and A. Wolf. On the translation of procedures to finite machines: Abstraction allows a clean proof. In G. Smolka, editor, *Programming Languages and Systems, Proceedings of the ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 290 – 304. Springer-Verlag, 2000.
64. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
65. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. Wiley, 1992.
66. 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.
67. D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8:111–182, 1995.
68. D. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78, 1969.
69. H. Pfeifer et. al. Supporting refinement calculus proofs in PVS. Verifix Working Paper [Verifix/Ulm 3.0], University of Ulm, Germany, 1996.
70. G. D. Plotkin. A powerdomain construction. *SIAM J. Computation*, 5:452–487, 1976.
71. G. D. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Comput. Sci. Dept., 1981.
72. 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.
73. A. Sampaio. *An Algebraic Approach To Compiler Design*. PhD thesis, Oxford University Computing Laboratory, 1993.
74. G. Schmidt and T. Ströhlein. *Relations and Graphs*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1991.
75. D. Scott. Outline of a mathematical theory of computation. In *4th. Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.
76. S. Sippu and E. Soisalon-Soininen. *Parsing Theory Vol. I*. Springer-Verlag, 1988.
77. M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.
78. A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
79. A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.
80. R. Verhoeven and R. Backhouse. Interfacing Program Construction and Verification. In J. Wing, J. Woodcock, and J. Davies, editors, *FM '99 – Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1128–1146. Springer-Verlag, 1999.
81. B. von Karger. Temporal algebra, 1997. Habilitation Thesis, Technical Faculty of the Christian-Albrechts-University, Kiel, Germany.
82. W. M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, 1984.
83. M. Wand. A characterization of weakest preconditions. *JCSS*, 15:209–212, 1977.
84. R. Wilhelm and D. Maurer. *Übersetzerbau (in German)*. Springer-Verlag, 1992.
85. A. Wolf. An Exercise in Compiler Verification Revisited – Preserving Partial Correctness. Verifix report Verifix/CAU/5.1, University of Kiel, Kiel, Germany, 1999.

Index