PROGRESSIVE PROGRAM REASONING

RĂZVAN VOICU

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgments

First, I would like to thank my supervisor, Joxan Jaffar, who has been a constant source of inspiration and encouragement, and the warmest of friends. In addition to the many technical things I've learned from Joxan, I have especially valued his eye for both elegance and relevance, as well as the enormous energy, focus, and precision he brings to everything he does. I strive to emulate Joxan in these ways, and many more.

I have been fortunate enough to have several other mentors along the way. I am greatly indebted to Nevin Heintze, who has been a very kind host, research supervisor, and friend during my visits to Bell Labs. I am also very grateful to Wei-Ngan Chin, Siau-Cheng Khoo, and Roland Yap, whose guidance, encouragement and feedback have been particularly important.

I would also like to thank Andrew Santosa, Rafael Ramirez, Hugh Anderson, Hui Wu, Florin Crăciun, Corneliu Popeea, Mihai Asăvoae, Măriuca Gheorghiţă and Anton Klechenov for being great friends and colleagues throughout the years, and contributing to a fun and exciting environment, in and out of office.

Finally, my deepest heartfelt gratitude goes to Diana and Tudor, and to my parents, Monica and Mihail, for their unconditional love and support, and for bearing with a busy and grumpy husband, father and son for so many years. This work is dedicated to them.

Contents

Summary

This work presents a general program reasoning framework that is

- able to express a wide range of properties of programs, including but not limited to safety, liveness, and temporal properties;
- compositional, in a Hoare-logic, assume-guarantee fashion;
- allows the use of assertions;
- has potential for combining automated methods with user-provided information;
- incremental, in the sense that it is not needed for the reasoning process to complete in order to derive useful information about the program; rather, useful information can be derived after every reasoning step.

We base our work on a semantics, called the *progressive semantics*, which captures an abstraction of the sequence of events in a program, as opposed to a flat set of states. Program properties are descriptions, or rather, approximations of this semantics. Our approximation scheme uses a set of sets of states (or family) to approximate a set of states. Thus, an approximation of the progressive semantics at a program point is a sequence of families.

To be able to reason symbolically about program behavior, we define assertion languages whose formulas are interpreted as sequences of families. Finally, we present the entire framework centered around a propagation operator, computing the strongest-postcondition of assertions across program points. An underlying philosophy is that assertions are freely associated with any program points. Further, an assertion is not just a specification of properties which are to be proved, but an assertion may also be used as an *assumption* in order to prove other assertions, including itself. We thus introduce a notion of *conditional correctness*, which makes the entire framework incremental. Each assertion is initially assumed, and may, in the verification process, become proved. In the end, a proved program is correct on the proviso that its (hopefully few) unproven assertions are correct. Thus, we do not need to wait for the reasoning process to complete; the fact that the already proved assertions are conditionally correct is useful information available before the completion of the program reasoning procedure.

List of Figures

List of Symbols and Notations

Symbol	Description	Page
P	Annotated program	59
E	Program Expression	59
C	Program Constraint	59
$\mathcal{F}$	Family Description Formula	167
$\mathcal{K}$	Symbolic Configuration	211
Σ	Set of environments	69
σ	Environment	69
Ψ	Indexed set	89
Σ_0	Start (initial) environment	75
θ	Trace	75
Θ	Set of traces	75
Γ	Set of configurations	91
ϵ	The empty trace	75
IdxEnv	Set of indexed sets	89
Idx	Set of all indexed sets of environments	69
IdxSingleton	Set of indexed singletons	89
Var	Set of program variables	59
AVar	Set of array variables	59

Symbol	Description	Page
SVar	Set of scalar variables	59
States	Set of all states	69
Values	Set of program values (integers,arrays of integers)	69
Expr	Set of program expressions	59
SExpr	Set of scalar expressions	59
AExpr	Set of array expressions	59
Constr	Set of program constraints	59
Bool	Set of truth values	59
AStmt	Set of annotated statements	59
AProg	Set of annotated programs	59
Labels	Set of labels used to identify program points	64
Singletons(P)	The set of singleton configurations with skeleton P	121
Progressive(P)	The set of progressive configurations with skeleton P	121
Collecting(P)	The set of collective configurations with skeleton P	134
Env	Set of environments	69
Fam	Set of families	149
$\mathbb{N}$	Natural numbers	79
$\langle l, \mathcal{F} \rangle$	Program point annotation	59
$K_1 \preccurlyeq K_2$	Partial order on configurations	91
$s_1 \xrightarrow{P} s_2$	Transition relation	68
$s_1 \xrightarrow{P}{}^{*} s_2$	Reflexive and transitive closure of $\xrightarrow{P}$	82
$\langle l, \sigma \rangle$	A state is a pair of a label and an environment	68
$E(\sigma)$	Value of expression E in env. σ	70
$\sigma \models C$	Constraint E is true in env. σ	70

Symbol	**Description**	**Page**
$\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$	Galois connection	40
$\sigma[x \mapsto E]$	Substitution	296
$l_1 \circledcirc l_2$	Labels at the same level	95
$l_1 \nearrow l_2$	Label l_1 is at a deeper level than label l_2	95
$l_1 \circledcirc l_2$	Label l_1 is shallower than label l_2	95
$seq(K_1, K_2)$	Sequencing operator for configurations	103
$extr(K, \mu_0)$	Extraction operator for configurations	103
$\|P\|$	The skeleton of a program P	90
$\vec{T}_P$	Trace progress operator	78
$\vec{P}$	The traces of a program P	75
$\overset{\Sigma_0\rightarrow}{P}$	Traces of P starting from the start environment Σ_0	75
T_P	Transfer function of P	84
T^p	Progress operator	122
$\widehat{T}$	Family progress operator	154
$lub(X)$	Least Upper Bound	37
$glb(X)$	Greatest Lower Bound	37
$lfp(T_P)$	Least Fixed Point of an operator	37
$before(\Psi_1, \Psi_2)$	"before" operator for indexed sets	106
$assign(x, E, \Psi)$	"assign" operator for indexed sets	106
$filter(C, \Psi)$	"filter" operator for indexed sets	106
$\widehat{before}(\Phi_1, \Phi_2)$	Before operator for families	154
$\widehat{assign}(x, E, \Phi)$	Assign operator for families	154
$\widehat{filter}(C, \Phi)$	Filter operator for families	154
$\Phi_1 \widehat{\cup} \Phi_2$	Union operator for families	154

Symbol	Description	Page
$\mathit{Before}(\mathcal{F}_1, \mathcal{F}_2)$	Before operator for formulas	154
$\mathit{Seq}(\mathcal{F}_1, \mathcal{F}_2, \ldots)$	Sequencing operator for formulas	167
$\mathit{Assign}(x, E, \Phi)$	Assign operator for formulas	167
$\mathit{Filter}(C, \Phi)$	Filter operator for formulas	167
$\mathit{Collect}$	Collect for formulas	167
$\sqcup$	Join operator for Formulas	167
$\sqcap$	Meet operator for Formulas	167
$\mathcal{T}(\mathcal{K})$	Symbolic propagator	213
$\xrightarrow{P\swarrow}$	Partial transition relation	70
$\xrightarrow{P\nearrow}$	Partial transition relation	70
$\xrightarrow{P\searrow}$	Partial transition relation	70
$\xrightarrow{P\nwarrow}$	Partial transition relation	70
$\xrightarrow{P\searrow}$	Partial transition relation	70
$\xrightarrow{P\nwarrow}$	Partial transition relation	70
$\xrightarrow{P\curvearrowright}$	Partial transition relation	70
$\xrightarrow{P\circlearrowleft}$	Partial transition relation	70
$\underbrace{t}_{P}$	Trace segment that is a trace of a component	76
K	Generic configuration	90
$\vec{K}$	Singleton configuration	90
$\overline{K}$	Collecting Configuration	90
$\hat{K}$	Family Configuration	149
$\lambda\langle\mu\rangle\,.\,\{\sigma \mid \sigma(x) > 0\}$	Indexed set specification	89
$[\![\mathcal{F}]\!]$	Interpretation of a family description formula	167
$\mathit{assert}(\mathcal{F})$	Assertion	224

Symbol	**Description**	**Page**
$assume(\mathcal{F})$	Assume notation	225
$replace(\mathcal{F}, l, \Lambda)$	Assertion replacement	226

Part I

Background

Chapter 1

Introduction

Good software is difficult to produce. This contradicts expectations, for building software requires no large factories or furnaces, ore or acres. It consumes no rare, irreplaceable materials, and generates no irreducible waste. It requires no physical agility or grace, and can be made in any locale. What good software does require, it demands of the intelligence and character of the person who makes it. These demands include patience, perseverance, care, craftsmanship, attention to detail, and a streak of the detective, for hunting down errors. Perhaps the most central is an ability to solve problems logically, to resolve incomplete specifications to consistent, effective designs, to translate nebulous descriptions of a program's purpose to definite detailed algorithms. Finally, software remains lifeless and mundane without a well-crafted dose of the artistic and creative.

Large software systems often have many levels of abstraction. Such depth of hierarchical structure implies an enormous burden of understanding. In fact, even the most senior programmers of large software systems cannot possibly know all the details of every part, but rely on others to understand each particular small area. Given that creating software is a human activity, errors occur. What is

surprising is how difficult these errors often are to even detect, let alone isolate, identify, and correct. Software systems typically pass through hundreds of tests of their performance without flaw, only to fail unexpectedly in the field given some unfortunate combination of circumstances. Even the most diligent and faithful applications of rigorous disciplines of testing only mitigate this problem. The core remains, as expressed by Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72] It is a fact that virtually every major software system that is released or sold is, not merely suspected, but in fact guaranteed to contain errors. This degree of unsoundness would be considered unacceptable in most other fields. It is tolerated in software because there is no apparent alternative. The resulting erroneous software is justified as being "good enough," giving correct answers "most of the time," and the occasional collapses of the system are shrugged off as inevitable lapses that must be endured. Virtually every piece of software that is sold for a personal computer contains a disclaimer of any particular performance at all. This means that the customer must hope and pray that the software performs as advertised, for he has no firm assurance at all. This lack of responsibility is not tolerated in most other fields of science or business. It is tolerated here because it is, for all practical purposes, impossible to actually create perfect software of the size and complexity desired, using the current technology of testing to detect errors. There is a reason why testing is inadequate. Fundamentally, testing examines a piece of software as a "black box[1]," subjecting it to various external stimuli, and observing its responses. These responses are then compared to what the tester expected, and any variation is investigated. Testing depends solely on what is externally visible. This approach treats the piece of soft-

[1] *White box* testing is also often used in practice; this is a technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data.

ware as a mysterious locked chest, impenetrable and opaque to any deeper vision or understanding of its internal behavior. A good tester does examine the software and study its structure in order to design his test cases, so as to test internal paths, and check circumstances around boundary cases. But even with some knowledge of the internal structure, it is very difficult in many cases to list a sufficient set of cases that will exhaustively test all paths through the software, or all combinations of circumstances in which the software will be expected to function.

In truth, though, this behavioral approach is foreign to most real systems in physics. Nearly all physical systems may be understood and analyzed in terms of their component parts. It is far more natural to examine systems in detail, by investigating their internal structure and organization, to watch their internal processes and interrelationships, and to derive from that observation a deep understanding of the "heart" of the system. Here each component may be studied to some degree as an entity unto itself, existing within an environment which is the rest of the system. This is essentially the "divide and conquer" strategy applied to understanding systems, and it has the advantage that the part is usually simpler than the whole. If a particular component is still too complex to permit immediate understanding, it may be itself analyzed as being made up of other smaller pieces, and the process recurses in a natural way.

This concept was recognized by Floyd, Hoare, Dijkstra, and others, beginning in 1969, and a wealth of alternative techniques to testing is currently in the process of being fashioned by the computing community. Among these approaches we mention program correctness, program analysis, and model checking. They are concerned with analyzing a program down to the smallest element, and then synthesizing an understanding of the entire program. In general, all these approaches use the means of mathematical proof in order to discover and guarantee properties of programs.

This is why we have coined the term *program reasoning* to refer to this general category of techniques.

As opposed to testing, reasoning can trace every path through a system, and consider every possible combination of circumstances, and be certain that nothing has been left out. This is possible because the method relies on mathematical methods of proof to assure the completeness and correctness of every step. What is actually achieved by reasoning is a mathematical proof that the program being studied satisfies its specification. If the specification is complete and correct, then the program is guaranteed to perform correctly as well.

However, the claims of the benefits of program reasoning need to be tempered with the realization that substantially what is accomplished may be considered an exercise in redundancy. The proof shows that the specification and the program, two forms of representing the same system, are consistent with each other. But deriving a complete and correct formal specification for a problem from the vague and nuanced words of an English description is a difficult and uncertain process itself. If the formal specification arrived at is not what was truly intended, then the entire proof activity does not accomplish anything of worth. In fact, it may have the negative effect of giving a false sense of certainty to the user's expectations of how the program will perform. It is important, therefore, to remember that what program reasoning accomplishes is limited in its scope, to proving the consistency of a program with its specification[2].

Within that scope, however, program reasoning becomes more than redundancy when the specification is an abstract, less detailed statement than the program. Usually the specification as given describes only the external behavior of the program. In one sense, the proof maps the external specification down through the

[2]A similar argument can be made about the consistency of testing.

structure of the program to the elements that must combine to support each requirement. In another sense, the proof is good engineering, like installing steel reinforcement within a large concrete structure. The proof spins a single thread through every line of code, but this single thread is far stronger than steel; it has the infinite strength of logical truth. Clearly this greatly increases one's confidence in the finished product.

The theory for creating these proofs of program correctness has been developed and applied to sample programs. It has been found that for even moderately sized programs, the proofs are often long and involved, and full of complex details. This raises the possibility of errors occurring in the proof itself, and brings into question credibility. This situation naturally calls for automation. Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step, and ensures its soundness.

Program reasoning holds the promise in theory of enabling the creation of software with qualitatively superior reliability than current techniques. There is the potential to forever eliminate entire categories of errors, protecting against the vast majority of run-time errors. However, program reasoning has not become widely used in practice, because it is difficult and complex, and requires special training and ability. The techniques and tools that are presented here are still far from being a usable methodology for the everyday reasoning of general applications. The mathematical sophistication required is high, the proof systems are complex, and the tools are only prototypes. The results of this dissertation point the direction to computer support of this difficult process that make it more effective and secure. Another approach than testing is clearly needed. If we are to build larger and deeper structures of software, we need a way to ensure the soundness of our construction, or else, inevitably, the entire edifice will collapse, buried under the weight of its

internal inconsistencies and contradictions.

1.1 Program Reasoning

Throughout this work, we coin "program reasoning" as a general term to stand for a wealth of techniques and approaches that are concerned with either certifying that certain desirable properties hold, or automatically deriving such properties for sequential programs. In either case, the properties of interest are guaranteed to hold, as opposed to software testing, which offers no guarantees about the program being tested.

When talking about program reasoning we have to consider two main aspects: the desirable properties we wish programs to have, and the techniques involved in certifying or detecting such properties. In what follows, we shall briefly turn our attention to each of these aspects.

Before defining what program properties are, we first need to understand what we mean by "program execution", or "program behavior". These notions are usually specified formally by means of a program semantics. Program semantics is a mathematical instrument that assigns a more or less abstract meaning to a program of a given programming language. By saying that a program has a certain property $\mathcal{P}$, we actually understand that the meaning of the "program", as defined by the semantics at hand, has the property $\mathcal{P}$. The classic program reasoning approaches would focus on the following categories of properties:

- *partial correctness*, that is, properties that state that whenever a result is delivered, it is correct;
- *termination*, which states that the execution of a program reaches its final program point;

- *non-failure*, which states that undesirable properties, like segmentation violation, do not happen.

In recent years, however, the uses of sequential programs go beyond computationally intensive problems that exhibit an input-compute-output behavior. The increasingly wider use of embedded systems, and the need to certify sequential programs that operate under strict behavioral policies have placed new requirements on program reasoning methods. For example, software drivers operate in a concurrent environment and need to exclusively acquire resources (e.g. semaphores) that must be released eventually. Hence, a common requirement for the certification of software drivers is that all acquired resources are eventually released. Another example is a program on a system with a limited amount of memory cache, for which we want to prove that the set of values assigned to a pointer does not exceed a certain size. Such a property would ensure a low frequency of cache misses and a higher execution speed. It is clear that such properties do not fit into the classic range of properties listed above.

In order to deal with such requirements, we adopt a more general view of what the *desirable properties* of a program are. Throughout this work we shall distinguish between *safety* and *liveness* properties [3]. While this categorization does not cover all the types of properties that we shall encounter throughout this work, the distinction is important from a theoretical point of view, since safety and liveness properties require different kinds of reasoning techniques. One of the contributions of the present work is a unified framework that provides a uniform treatment of safety, liveness, and sequence-based properties, that is, properties that capture (an abstraction of) the sequence of events occurring during the execution of the program

[3] The definition of safety and liveness properties are borrowed from the concurrent programming community.

— we call such properties *progressive properties*.

There is a wealth of methods and techniques for proving or discovering properties of programs. The area of program reasoning has started with the seminal work of Floyd and Hoare, who have devised a calculus for proving partial correctness properties of programs. This calculus adopted the input-compute-output view of software, but had the advantage of being compositional, in an assume-guarantee fashion. The calculus has later been extended to handle termination, and has been the basis for propagation-based reasoning algorithms, which would operate either in a forward manner (strongest postcondition propagation), or in a backward manner (weakest precondition propagation). The power of the Hoare calculus extends to proving safety properties; it requires user-provided assertions and invariants, which makes automation rather difficult to achieve.

Verification condition generators represent an attempt to automate program correctness proofs as much as possible. They are tools that construct significant parts of the correctness proofs of a program, outputting a set of lemmas called verification conditions, as a remainder that is left for the user to prove. The truth of these is intended to imply the correctness of the program. However, most verification condition generators that have been built so far have not themselves been verified, making that support unreliable.

Another program reasoning approach is program analysis, which has been formalized in a semantics driven manner in the abstract interpretation framework. As a program analysis methodology, abstract interpretation relies on simulating the execution of the program on an abstract domain which is Galois connected with the concrete semantic domain. As a result, a set of concrete states of the program can be approximated by an abstract state, and the execution of the program in the concrete domain can be approximated by an abstract execution in the abstract domain.

If the abstract domain is a finite height lattice, the abstract computation will reach a fixpoint in a finite amount of time, thus providing a conservative approximation of the set of concrete states that occur during the execution of the program. Since the approximation specifies a superset of the semantics, it acts as an upper bound, and thus becomes a guaranteed property. Program analysis has the advantage of being completely automatic, but it lacks the flexibility of the Hoare-calculus-based methods, where the user can help along by specifying assertions and invariants. As a result, the outcome of a program analysis method is, in general, not necessarily "interesting". Moreover, it is usually difficult to produce analyses that are compositional in nature. In its classic form, abstract interpretation can only discover safety properties; however, this framework has been extended to handle termination properties and debugging.

A recent development in program reasoning is the predicate abstraction framework. This framework is tightly connected with abstract interpretation, in the sense that predicate abstraction uses as an abstract domain a set of user defined predicates that depend on program variables (for example, $x < y$, could be such a predicate, where x and y are program variables). However, the abstraction will be performed by specializing the program at hand into a (possibly non-deterministic) boolean program, that is, a program whose variables can only have the values *true*, *false*, and *undefined*. Intuitively, each variable in the boolean program corresponds to a user-defined predicate in the abstract domain. The execution of the boolean program would reach a fixpoint in finite time. There are two important advantages to this framework. First, it exhibits a high level of flexibility by allowing the user to choose the abstract domain. Second, since all execution paths are finite, the execution of the boolean program could record all abstract traces and discover properties about the *sequences of events* generated by the execution of the concrete program

(an example of such a property would be that x is assigned 1 before y is less than 2 for all possible executions of a given program).

Model checking is a verification technique that has been applied with tremendous success to hardware and other finite state systems in recent years. It has several advantages over classic program verification methods, the most important being that it is completely automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the specification is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The procedure is also quite fast and can check partial specifications. Yet, when it comes to verifying software systems, which are (at least theoretically) infinite state systems, the restriction to a finite state space that model checking has seems to be a major disadvantage. In this case, abstraction techniques can be used to produce a finite state approximation of the system[4]. The downside of using abstraction is that spurious errors will be detected as a result of the approximation. To deal with this, recent techniques have been developed to refine the abstraction mechanism on the fly to help distinguish between spurious and real errors.

1.2 This Work

Each of the program reasoning approaches described in the previous section has specific trade-offs between the level of automation, the compositionality, and the

[4]Abstraction is also a useful technique for reducing the state explosion of concurrent finite systems.

complexity of program properties that can be derived. The motivation for this work has been to create a general, unifying program reasoning framework, that retains as much as possible of the desirable features of the approaches mentioned above. In our view, these features are:

- Ability to express a wide range of properties of programs, including but not limited to safety, liveness, and behavioral properties (i.e. properties related to the sequence of events or states occurring during the execution of the program.)
- Compositionality, in a Hoare-logic, assume-guarantee fashion.
- Ability to use assertions, as a means of allowing the user to guide the proof along.
- Potential for combining automated methods with user-provided information.
- Incrementality, in the sense that it is not needed for the reasoning process to complete in order to derive useful information about the program; rather, useful information can be derived after every reasoning step.

At this point, a short comment on reasoning about behavior is in order. It has been argued [Bur72b, Lam77] that all the properties one would find of interest about a sequential program can be proved by adding new variables, whose values would capture an abstraction of the program execution's history, and then prove some safety property of the program, and also termination. This may lead to thinking that reasoning about program behavior is only useful in a concurrent/reactive/real-time setting. While we agree that most sequential programs that are *intended to run in a sequential environment* exhibit an input-compute-output-and-terminate behavior that justifies the above statement, we would like to emphasize that we

often need to reason about sequential programs that *were intended to run* in a non-sequential setting. Such situations arise when, by applying compositional proof methods [MC81, Jon83] to a concurrent program, the process of reasoning about the entire program is decomposed into behavioral proof obligations for its sequential components. Therefore, the ability to reason (in a compositional manner) about a sequential program's behavior is, in our opinion, particularly important, and has been included as one of the desirable properties of our framework.

The object of this thesis is the formalization of the type of reasoning presented in this example, and the rest of this section is devoted to outlining this process. Lets us first take a look at some of the shortcomings of classic program reasoning approaches, and understand why liveness and termination cannot be handled in a straightforward manner. Reasoning about program behavior relies on two important steps:

- Defining "program behavior" as a compositional semantics of the language at hand (the typical way of defining the behavior of a program is via the trace semantics [Sch98a, Col96, CL96], which is not compositional.)

- Defining the meaning (interpretation) of program properties we are interested in as approximations of the semantics.

For the reasoning process to expose the desired property of a program, we need that the program semantics capture a not-too-abstract level of information, and that the means of approximation carry that level of information through.

Classic program reasoning approaches rely on the collecting semantics as a definition of program behavior. The collecting semantics represents the set of states that occur during the execution of a program. In this setting, program properties are expressed as formulas whose interpretation are sets of states as well. Usually,

we say that a program has a certain property if the collecting semantics of the program is a subset of the interpretation of the corresponding formula. What usually happens is that the collecting semantics CS is unknown and the interpretation of the property at hand acts as an upper bound for CS. In other words, any subset of the interpretation, including the empty set, is a possible value of the semantics. For this reason, we cannot infer any liveness or termination information from a collecting-semantics-, subset-approximation-based type of reasoning. Indeed, since the empty set is a possible value, it is possible that the program does not perform any computation at all!

In contrast to this approach, we base our work on a less abstract semantics and a more flexible approximation scheme. Our semantics, called the *progressive semantics*, captures an abstraction of the sequence of events in a program, as opposed to a flat set of states. Moreover, our approximation scheme uses a set of sets of states (or family) to approximate a set of states. Given a set of states S, denote by S' one of its supersets. The set S' should be a valid approximation of S in a subset-based approximation scheme.

Our more general setting uses a family $F \subseteq 2^{S'}$. We say that F approximates S if $S \in F$. This is a more flexible scheme, as it allows the choice of restricting the range of possible values for S. In particular, we may choose an approximation F such that $\emptyset \notin F$. In this case, F may not specify precisely the computation the program performs, but it at least specifies that the program performs "some" computation (as opposed to no computation at all.)

The progressive semantics attaches some sort of sequence of sets to each program point[5], and each set in the sequence shall be approximated by a family. Thus, an approximation of the progressive semantics at a program point is a sequence

[5]We call such sequence of sets an *indexed set*

of families. The next step, is to define assertion languages whose formulas are interpreted as families. At this point, we have the means to reason symbolically about program behavior, and develop a program reasoning framework that treats liveness and safety properties in a unified manner.

Finally, we present the entire framework centered around a propagation operator, computing the strongest-postcondition of assertions across program points, and various correctness criteria, thus forming the foundations of the entire program reasoning calculus. An underlying philosophy is that assertions are freely associated with any program points. Further, an assertion is not just a specification of properties which are to be proved, but an assertion may also be used as an *assumption* in order to prove other assertions, including itself. We thus introduce a notion of *conditional correctness*, which makes the entire framework incremental. Each assertion is initially assumed, and may, in the verification process, become proved. In the end, a proved program is correct on the proviso that its (hopefully few) unproven assertions are correct. Thus, we do not need to wait for the reasoning process to complete; the fact that the already proved assertions are conditionally correct is useful information available before the completion of the program reasoning procedure.

1.3 Introductory Examples

In this section, we provide a motivation to our work with the aid of two examples, one that proves the termination of a program and illustrates the basic principle of reasoning within our framework, and a second one, which proves behavioral properties. Consider first the program fragment given in Figure 1.2, which is part of a larger program computing the greatest common divisor of two numbers. Since

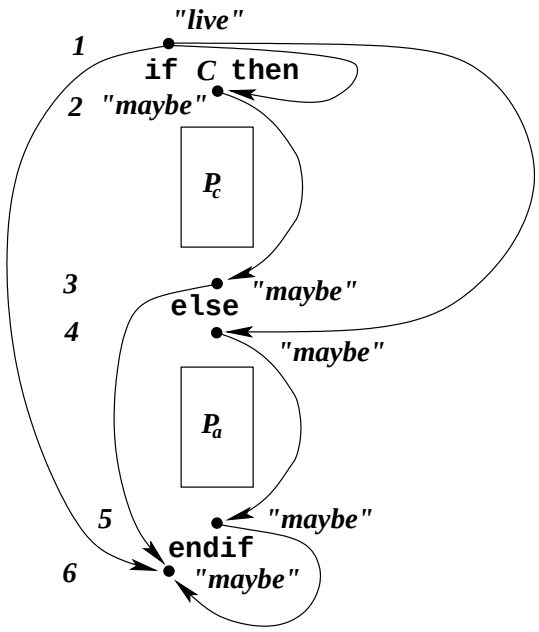


Figure 1.1: Liveness Information Propagated through an If Statement

it is well-known that Euclid's algorithm terminates, we would expect that once the first program point of the fragment is reached, the last program point of the fragment shall be eventually reached as well. This example discusses the challenges of inferring this information formally, and presents a solution. Based on this, we then discuss the formal road-map that we follow throughout this work.

Guaranteeing that once a program point has been reached, another shall be reached eventually is akin to proving total correctness in the classic program verification setting. In our work, however, we shall focus, among other things, on compositional aspects of program reasoning, and for this reason we see each piece of code as a fragment that could be nested within a larger program, and we aim at providing "reasonings" that are generic in that context. To this purpose, we shall prefer the term *liveness of a program point* as a means of expressing the certainty

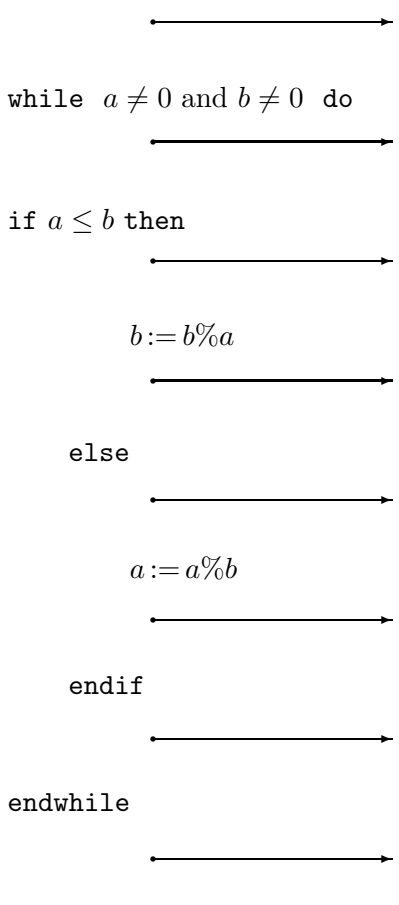
of a program point being reached.

The program uses Euclid's algorithm to compute the greatest common divisor of two natural numbers. The informal argument for the liveness of the last point in the program is that at each iteration through the loop, the value $\max(a, b)$ grows smaller and smaller, and therefore either of the variables a or b will hit the value 0 after same finite amount of time. To see that the value $\max(a, b)$ grows smaller and smaller, we have to look at the `if` statement in the program fragment. Indeed, the variable with the larger value is assigned the remainder of dividing the larger value by the smaller value, and therefore what was the smaller value before the `if` statement becomes the larger of the two values after its execution. Since neither of the branches of the `if` statement contains loops, there is no doubt that once the program point at the beginning of the `if` statement is reached, the program point at the end of the `if` statement will eventually be reached as well. While this intuition is very simple and straightforward, when reasoning formally about a program in a semantic-based manner, carrying liveness information through an `if` statement may not be equally straightforward.

Before explaining the program in Figure 1.2, we shall digress into analyzing a few aspects of reasoning about liveness. Program reasoning frameworks are typically based on a semantics that is defined hierarchically and compositionally, that is, in a manner in which the meaning of the program is made up from the meaning of its components, while the meanings of two distinct components are completely independent of each other. Assume now that we have a semantics that carries through liveness information, and imagine a scenario where we reason about an `if` statement. Based on the semantics at hand, our reasoning framework should be able to infer for a program point information of the form "live", "dead" or "maybe". This setting is depicted in Figure 1.1. Assume we are certain to reach

program point 1. Depending on the state in which this happens, execution could move either to program point 2 or program point 4. We are not certain to reach either of these two program points, and therefore the liveness information that could be inferred is "maybe" at best.

Since the semantic meanings of the program fragments P_c and P_a are independent, propagating liveness information through P_c and P_a shall produce a "maybe" for both program points 3 and 5. Neither of these two program points are guaranteed to be reached, and as a result, the only liveness information we can propagate to program point 6 is "maybe", which contradicts the intuition that program point 6 is definitely reached once program point 1 is reached. This argument shows that one of the challenges in reasoning about liveness is correlating information coming from independent program fragments. A solution to this problem is presented in the example given in Figure 1.2. The table beside the program in Figure 1.2 annotates each program point with a formula which states a property that holds each time the respective program point is reached. In contrast with the classic program reasoning setting, where properties are specified as flat, first-order formulas on program variables, our example employs parameterized formulas that provide a certain structure to the collection of states occurring at that program point, and make the correlation of liveness information between independent program fragments possible. The parameters to our formulas are natural numbers and provide information about the order in which states occur at a program point or, more abstractly, provide a way of enumerating the states occurring at a program point. Lets take, for example the formula attached to program point 1. This formula has two parameters, ν_1 and ν_2. For each pair of values ν_1 and ν_2 we get sets of possible values for variables a and b. This is a way of saying that the values of a and b specific to a pair of values for ν_1 and ν_2 occur separately, or in isolation of other values of a and b, specific to



1	$\lambda\langle\nu_1\nu_2\rangle.(\ (\nu_1\%2 = 0 \rightarrow 0 \leq a \leq b) \wedge (\nu_1\%2 \neq 0 \rightarrow a > b \geq 0) \wedge$ $(\max(a,b) \leq \nu_2)\ )^*$
2	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ ((\nu_1+\nu_3)\%2 = 0 \rightarrow 0 < a \leq b) \wedge$ $((\nu_1+\nu_3)\%2 \neq 0 \rightarrow a > b > 0) \wedge (\max(a,b) \leq (\nu_2 - \nu_3))\ )^*$
3	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ (\nu_1+\nu_3)\%2 = 0 \wedge 0 < a \leq b \leq \nu_2 - \nu_3\ )^*$
4	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ (\nu_1+\nu_3)\%2 = 0 \wedge (0 \leq b < a \leq \nu_2 - \nu_3 - 1)\ )^*$
5	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ (\nu_1+\nu_3)\%2 \neq 0 \wedge 0 < b < a \leq \nu_2 - \nu_3\ )^*$
6	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ (\nu_1+\nu_3)\%2 \neq 0 \wedge 0 \leq a < b \leq \nu_2 - \nu_3 - 1\ )^*$
7	$\lambda\langle\nu_1\nu_2\nu_3\rangle.(\ ((\nu_1+\nu_3)\%2 = 0 \rightarrow 0 \leq b < a \leq \nu_2 - \nu_3 - 1) \wedge$ $((\nu_1+\nu_3)\%2 \neq 0 \rightarrow 0 \leq a < b \leq \nu_2 - \nu_3 - 1)\ )^*$
8	$\lambda\langle\nu_1\nu_2\rangle.(\ a = 0 \vee b = 0)^*$

Figure 1.2: Greatest Common Divisor Program

different pairs of values of ν_1 and ν_2. A more concrete interpretation of parameters ν_1 and ν_2 would be that the program fragment is in fact part of a larger program, and is nested inside two `while` loops. Then ν_1, could be regarded as the number of iterations in the outer loop, and ν_2 as the number of iterations in the inner loop. This intuition becomes more apparent in the formula attached to program point 2, which has three parameters. Program point 2 is inside a loop, and we regard ν_3 as a counter which is initialized to 0 before entering the loop, and is incremented every time around the loop. Such formulas, when used as invariants, allow the specification of more detailed properties. For example, this will help to carry liveness information through the `if` statements. As argued above, we cannot infer that program points 3 and 5 are definitely reached. However, using the ν_3 parameter, we can prove that when $a \leq b$ at the beginning of the program, program point 3 will definitely be reached during the even iterations through the `while` loop body, whereas program point 5 will be reached during the odd iterations. Propagating this information through the branches of the `if` statement, we get that program point 4 is reached during the even rounds through the loop and program point 6 is reached during the odd rounds. As a result, we can infer that program point 7, at the end of the `if` statement, is reached during all rounds. A similar argument can be made for the case when $a > b$ at the beginning of the program, with the odd and even rounds switching roles. Thus, we can correlate information collected from independent program fragments. In order for this to be achieved, we need to:

- Distinguish between two cases at the beginning of the program : either $a \leq b$, or $a > b$ — this is the role of parameter ν_1.

- Record the value $\max(a, b)$ at the beginning of the program, to be able to show that this value grows smaller with every iteration through the loop; this

is the role of parameter ν_2.

- Model the passage of time and record some abstraction of the sequence of events that happen during the execution of the while loop; this is the role of parameter ν_3 — we can show that $\max(a, b) \leq \nu_2 - \nu_3$, and that as ν_3 grows, $\max(a, b)$ grows smaller.

Using parameterized assertions is not enough to prove liveness. We need formulas that are able to express *how precise* our knowledge about liveness is. In this respect, we shall regard the "live" and "dead" information as more precise than the "maybe" information. To express the fact that we have precise knowledge about the liveness of a program point, we use formulas annotated by the "$*$" symbol. Liveness-wise, a starred formula has the meaning of either "live" or "dead", but not "maybe", whereas a non-starred formula always has the meaning of "maybe."

Our formulas are of the form $\lambda\langle\nu_1, \ldots, \nu_k\rangle.\varphi$, or $\lambda\langle\nu_1, \ldots, \nu_k\rangle.\varphi^*$, where φ is a first order formula in which variables $\nu_1, \ldots, \nu_k$, and program variables like a and b appear free. Whether starred formulas are interpreted as "live" or "dead" depends on the satisfiability of φ; for some values of the parameters $\nu_1, \ldots, \nu_k$ there may be a valuation assigning values to program variables that satisfies φ, for some other values of the parameters such valuation may not exist. For these values of the $\nu_1, \ldots, \nu_k$ parameters for which φ is satisfiable, the program point at hand is "live", whereas for those values of the parameters for which φ is unsatisfiable, the program point at hand is "dead."

We shall now take a look at some of the formulas attached to program points in our example, and explain their meaning and role in the liveness proof. The formula attached to program point 1 acts as a precondition to the entire program fragment — all other formulas are meaningful only if the formula at program point 1 has been

satisfied upon entry into the program fragment. For a given valuation of ν_1 and ν_2, this formula specifies a particular set of values for variables a and b. The intuition behind ν_1 and ν_2 is that all other formulas will be true of the current values of a and b for the same valuation[6]. On the other hand, as ν_1 and ν_2 sweep the entire set of natural numbers, there are no positive values of a and b left out. In other words, the formula at program point 1 specifies all natural numbers as possible values for a and b. The parameters ν_1 and ν_2 act as means of providing structure for the set of initial values of a and b. The formula at program point 1 is starred, and since the first order formula

$$(\nu_1 \% 2 = 0 \rightarrow a \leq b) \wedge (\nu_1 \% 2 \neq 0 \rightarrow a > b) \wedge (\max(a, b) \leq \nu_2)$$

is satisfiable, for each valuation assigning values to ν_1 and ν_2 we have that program point 1 is "live". This is in fact an assumption under which the entire reasoning process is performed. In order to explain our choice for structuring the initial values of a and b in this way, we need to take a look at the `if` statement in our program. For a fixed valuation of ν_1 and ν_2, the values of a and b throughout the entire run of the program are such that every time we are at program point 2, we know precisely whether program execution will advance to program point 3, or program point 5. Thus, we can infer that program point 3 is definitely reached in the even iterations through the `while` loop, whereas program point 5 is reached in the odd iterations. As argued above, this information can be propagated to program points 4 and 6, and then to program point 7, to show that program point 7 is "live". Had we not started with the initial values of a and b structured in the way they are, we would have only inferred "maybe" for program points 3, 4, 5, 6 and 7.

[6]The reader may wonder how straightforward such an intuition is. More detailed explanations on choosing parameters ν_1 and ν_2 are provided in Chapter 15 and in Section 16.2.

Looking at the formula attached to program point 2, we notice a new parameter ν_3. This parameter acts as a counter inside the loop — we can think of it as being initialized to 0 just before entering the loop, and then incremented by 1 every time around the loop. Based on this parameter, we can differentiate between even and odd iterations through the loop. The formula states that during the even iterations, if ν_1 is even as well, we have that $a \leq b$, while during the odd iterations we have $a > b$. Since the formula at program point 3 is starred, and its embedded first order formula

$$((\nu_1 + \nu_3)\%2 = 0 \rightarrow a \leq b) \wedge ((\nu_1 + \nu_3)\%2 \neq 0 \rightarrow a > b) \wedge (\max(a, b) \leq (\nu_2 - \nu_3))$$

if satisfiable for all valuations assigning values to ν_1, ν_2 and ν_3, it follows that program point 2 is reached in every iteration through the loop. The formula at program point 3 results from propagating the formula at program point 2 through the `if` condition. This formula is still starred. Taking a look at the first order formula embedded at program point 3.

$$(\nu_1 + \nu_3)\%2 = 0 \wedge a \leq b \leq \nu_2 - \nu_3$$

We see that it is satisfiable only for valuations of ν_1, ν_2 and ν_3 such that $(\nu_1 + \nu_3)$ is even. The star annotation means in this case that program point 3 is "live" only during the even iterations through the loop, and "dead" during the odd iterations. However, since we know exactly when the program point is "live" or "dead," this formula retains the "$*$" annotation.

The formula at program point 7 comes from bringing together the formulas at program points 4 and 6, since program point 4 is "live" on one parity of ν_3, while program point 6 is "live" on the other parity or ν_3, it follows that program point 7 is always "live". Propagation through the branches of the `if` statement has

also brought down the upper bound of $\max(a, b)$. As ν_3 increases, iteration after iteration, the upper bound on $\max(a, b)$ shall get closer and closer to 0.

Finally, we turn our attention to the formula at program point 8. This formula is produced by existentially quantifying the parameters ν_3 in the first order formula at program point 7, and then conjoining it with the negation of the while condition. As a result, for every valuation assigning values to ν_1 and ν_2, there exists a value of ν_3 such that

$$\begin{aligned}((\nu_1 + \nu_3)\%2 = 0 \rightarrow b < a \leq \nu_2 - \nu_3 - 1) \wedge \\ ((\nu_1 + \nu_3)\%2 \neq 0 \rightarrow a < b \leq \nu_2 - \nu_3 - 1) \\ \vee b = 0 \rightarrow a = 0 \vee b = 0.\end{aligned}$$

For this reason, the formula, at program point 8 retains its star annotation. We notice now that the formula at program point 8 is satisfiable for every valuation assigning values to ν_1 and ν_2, which entails that program point 8 is live.

Our second example uses Stenning's protocol [Ste76] to illustrate the fact that reasoning about the behavior of sequential programs can be useful in verifying concurrent systems. In Stenning's protocol a sender transmits a message to a receiver over unreliable channels. By "unreliable" we mean a channel that may lose, duplicate, or reorder packets, though if a packet is sent repeatedly over the channel, that packet cannot be lost forever. The purpose of the protocol is to achieve reliable communication, that is, to transmit *all* packets *exactly once*, such that they would reach the destination in the order in which they were transmitted.

The communication system has a sender and a receiver, both sender and receiver have interfaces to the unreliable channels. Each interface consists of two processes S_1, S_2, and R_1, R_2, respectively, each pair of processes being linked by a reliable channel. Reliable channels are represented by thin lines in Figure 1.3. For the sending interface, process S_1 receives a datum from the sender, and creates a packet

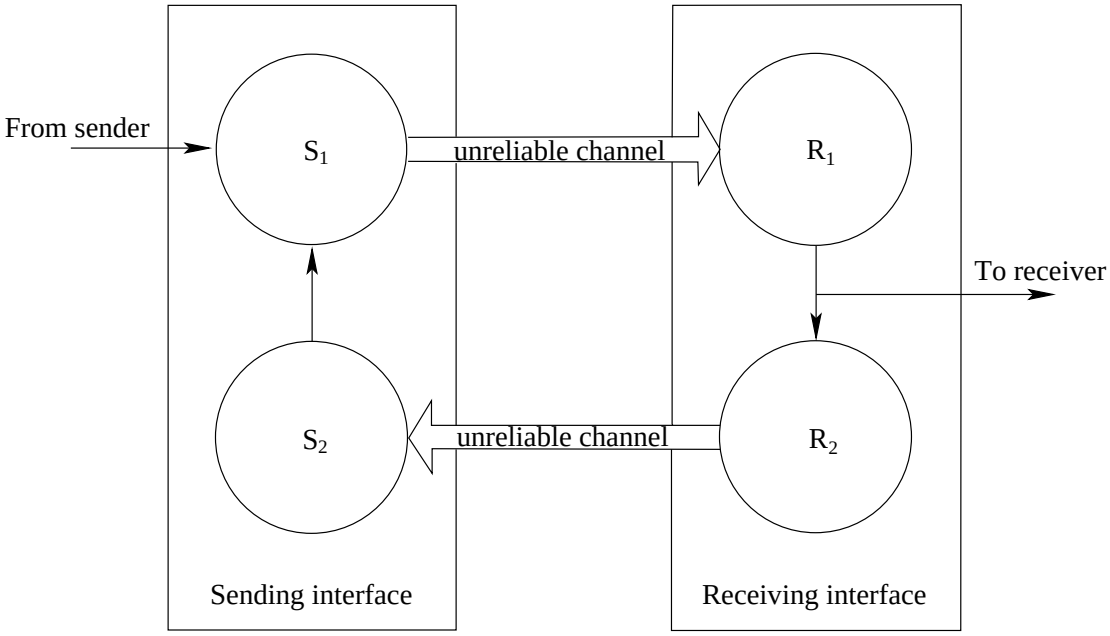


Figure 1.3: Communication Over Unreliable Channels Using Stenning's Protocol

by pairing it up with a tag, which is an integer, distinct for every packet that is sent. Then, S_1 sends the packet across the unreliable link until it receives confirmation from S_2 that the packet has arrived to its destination. The confirmation is in the form of the same message being received from S_2. Each new packet sent by S_1 will have a new tag, whose value is equal to the value of the previous tag plus 1. Thus, the tags establish the order in which the messages must be received at the destination. An unreliable channel cannot lose the same packet forever, and thus at least one of the packets with a given tag makes its way to R_1. Process R_1 reads packets from the unreliable channel, expecting for the "next" packet, that is a packet whose tag is one greater than the previously-received-and-not-discarded packet. Due to duplication and reordering, R_1 may discard packets repeatedly

```
⟨1⟩
  nextread := 0
⟨2⟩
  nextwrite := 0
⟨3⟩
  recvtag := −1
⟨4⟩
  exptag := 0
⟨5⟩
  while true do
    ⟨6⟩
      while recvtag ≠ exptag do
        ⟨7⟩
          recvtag := input[nextread]
        ⟨8⟩
          nextread := nextread + 1
        ⟨9⟩
      endwhile
    ⟨10⟩
      exptag := exptag + 1
    ⟨11⟩
      output[nextwrite] := recvtag
    ⟨12⟩
      nextwrite := nextwrite + 1
    ⟨13⟩
  endwhile
⟨14⟩
```

Figure 1.4: The Receiver Process in Stenning's Protocol

before receiving the "expected" one. Once an expected packet has been received, that packet shall be sent to the receiver, as well as to process R_2, which will confirm the receipt of the packet to the sender by repeatedly sending it over the unreliable channel until a new packet is received from R_1. Since the unreliable link cannot lose the same packet forever, the confirmation packet will eventually make its way to S_2, which will relay the confirmation to S_1. Upon receipt of confirmation for a given packet, S_1 can engage in sending the next packet. It is rather easy to prove

[Lyn96] that this protocol achieves reliable communication.

Our example shall focus only on the (sequential) code of process R_1 (in fact, a simplified version of R_1, where the datum to be sent is always nil.) The code is given in Figure 1.4. We note that this program fragment does not exhibit the input-compute-output-and-terminate behavior that is typical of sequential programs. The program fragment does not terminate, and input and output operations are spread throughout the entire execution of the program. However, the basic principles of verifying sequential code still do apply. As argued previously, one of the desirable properties of verifying (sequential or concurrent) programs is compositionality, and in the case of concurrent programs, this is achieved in an assume-guarantee fashion. That is, for each sequential process, we assume a pattern of interaction between processes, and we prove that, as long as all the other processes obey this pattern, the process at hand also does. In such a case, even though input and output operations occur throughout the entire execution of the program, we may assume that the desired sequences of inputs and outputs are known before the program execution starts, and then prove that the program indeed realizes the desired output for a given input. For this reason, without any loss of expressive power, input and output operations can be modeled as reading and writing elements from or into consecutive positions of input/output arrays, called *input* and *output* in our example, respectively. The program fragment starts with an expected tag of 0, and executes forever. The inner while loop reads a packet from the input channel (which is unreliable) and checks whether it has the expected tag. If it doesn't, then it discards the packet; otherwise, it outputs the packet (i.e. it sends it to R_2 via the reliable channel), and increases the expected tag by one. Using progressive program reasoning, we can prove that, under the assumption that the input array contains an infinite substring of all positive integers in increasing order, the output of the

program is the sequence of all positive integers in increasing order, which establishes the correctness of the program.

1.4 Outline of the Thesis

The thesis is structured into 16 chapters, divided into 4 parts. In Part I, Chapter 2 covers the basic concepts used throughout the thesis. Chapter 3 is a survey of related work. In Part II, we cover various programming language semantics. Chapter 4 introduces a simple programming language, and the notion of annotated program, which is a data structure that captures both the program, and the information derived in the process of reasoning about the program. In Chapter 5 we define the structural operational semantics of our language and based on this, we introduce the trace and collecting semantics. In Chapter 6, we show that semantics can be expressed as annotated programs, which we call configurations, and we show that this approach fosters compositionality. In Chapters 7 and 8 we provide the details of representing the trace and collecting semantics as configurations. Chapter 9 introduces the progressive semantics as a middle-ground between the trace and the collecting semantics, and Chapter 10, shows the semantics hierarchy of the trace, progressive, and collecting semantics. Part III discusses progressive reasoning. In Chapter 11, we introduce families of sets of states as a means of semantic approximation. Chapter 12, shows how a Hoare calculus can be built on top of the progressive semantics. Such a calculus is parameterized by a family description language, which is a formal-system-type language whose formulas are interpreted as families. Chapter 13, shows an example of a family description language, and the resulting progressive Hoare calculus. Chapter 14 introduces symbolic configuration as means of organizing the information produced throughout the program

reasoning process and shows how to perform strongest post-condition propagation. Chapter 15 introduces conditional reasoning as a means to realize an incremental and compositional program reasoning framework. In Part IV, Chapter 16 discusses future work and concludes.

Chapter 2

Underlying Technologies

During the past decade, a lot of progress has been made, both in methodological tools that enhance the intellectual ability of coping with complex software systems, and mechanical tools to help the programmer to reason about programs. Mechanical tools for program verification started by executing or simulating the program in as many environments as possible. However, debugging of compiled code or simulation of a model of the source program hardly scale up, and often offer a low coverage of dynamic program behavior. Formal program verification methods attempt to mechanically prove that program execution is correct in all specified environments. This includes program verification methods, program analysis, and model checking. Since program verification is undecidable, program reasoning methods are either partial or incomplete. The undecidability or complexity is always handled using some form of approximation. This means that the mechanical tool will sometimes suffer from practical time and space complexity limitations, rely on finiteness hypothesis, or provide only semi-algorithms, require user interaction, or be able to consider restricted forms of specification or programs only.

In this chapter we provide a succinct coverage of the underlying program rea-

soning technologies: program verification and abstract interpretation.

2.1 Program Verification

The area of program verification has started with the publication of [Flo67]. A formal foundation to this field has been given by the publication of Hoare logic in [Hoa69]. Hoare logic is a formal system for reasoning about formulas called Hoare triples, which are able to express the correctness of programs. The formal system consisted of a set of axioms for proving the partial correctness of programs constructed using assignments, sequential composition, conditional and `while` statements. Soundness and relative completeness proofs for this logic were given for the first time in [Coo81].

Hoare's approach has received a great deal of attention ever since its introduction, and has had a significant impact on methods both for designing and for verifying programs. It owes its success to three factors. Firstly, it is state-based, that is, characterizes programming constructs as state transformers, and therefore, applies in principle to every such construct. Secondly, it is syntax-directed, and can, therefore, also be regarded as a design calculus. Thirdly, it has a very simple way of characterizing sequential composition and iteration.

Hoare logics have been formulated for a wide range of, sometimes application-specific, languages, ranging from pattern matching to graphical languages, and with the ability to express properties ranging from fault tolerance to real time. Exhaustive reviews of this topic are presented in [Apt81, Dah92].

Closely related to Hoare logics are calculi for predicate transformers. In this approach, one characterizes the semantics of a programming language by functions mapping predicates to predicates. This approach was introduced by Dijkstra in

[Dij75].

A more compact way of expressing Hoare logic proofs are the proof outlines introduced in [Owi75], and formalized in [Apt83].

The set of rules for the classic (partial correctness) Hoare logic for a simple imperative language are given in Figure 2.1. Each formula of the form $\{\mathcal{F}\}\ \mathcal{P}\ \{\mathcal{F}'\}$ has three components: a precondition $\mathcal{F}$, a program fragment $\mathcal{P}$, and a postcondition $\mathcal{F}'$. The pre- and post-conditions are typically first order formulas in which program variables appear free. Then, these conditions are satisfied by a program state σ if by substituting the values of the program variables in a formula, the formula becomes true. The formula $\{\mathcal{F}\}\ \mathcal{P}\ \{\mathcal{F}'\}$ is called a *Hoare triple*, and has the following partial correctness interpretation: if the program fragment $\mathcal{P}$ is started in a state that satisfies $\mathcal{F}$, then, if and when the program fragment terminates, the terminating state of $\mathcal{P}$ satisfies $\mathcal{F}'$. This calculus makes it possible to verify *safety properties*, that is, properties that are guaranteed to be satisfied *if and when execution reaches that program point*. Safety properties do not guarantee that the program will perform any useful computation, since they cannot guarantee that a program point is reached at all. Safety properties, however, guarantee that "nothing bad" will happen, in the sense that the negation of the safety formula $\mathcal{F}$ cannot be satisfied by any state during the execution of the program.

In order to prove total correctness, the Hoare rules have been augmented such that termination of a program fragment could be inferred. The augmented rules are given in Figure 2.2. The while rule (T-WHILE) is augmented with measure t, whose value before the execution of the loop is equal to that of the logical variable z, which is a variable that does not appear in the program fragment at hand, and therefore cannot be modified during execution. One execution of the `while` loop body is guaranteed to decrease the measure. This is ensured by imposing the condition

$$\frac{\{\mathcal{F}_1\}\ \mathcal{P}\ \{\mathcal{F}_2\}}{\{\mathcal{F}'_1\}\ \mathcal{P}\ \{\mathcal{F}'_2\}} \qquad \begin{array}{l}\mathcal{F}'_1 \rightarrow \mathcal{F}_1\\ \mathcal{F}_2 \rightarrow \mathcal{F}'_2\end{array} \qquad \text{(P-CONSEQ)}$$

$$\frac{}{\{\mathcal{F}\}\ \texttt{skip}\ \{\mathcal{F}\}} \qquad \text{(P-SKIP)}$$

$$\frac{}{\{\mathcal{F}[E/x]\}\ x := E\ \{\mathcal{F}\}} \qquad \text{(P-ASSIGN)}$$

$$\frac{\{\mathcal{F}\}\ \mathcal{P}_1\ \{\mathcal{F}'\} \quad \{\mathcal{F}'\}\ \mathcal{P}_2\ \{\mathcal{F}''\}}{\{\mathcal{F}\}\ \mathcal{P}_1 \,;\, \mathcal{P}_2\ \{\mathcal{F}''\}} \qquad \text{(P-SEQ)}$$

$$\frac{\{\mathcal{F} \wedge C\}\ \mathcal{P}_c\ \{\mathcal{F}_1\} \quad \{\mathcal{F} \wedge \neg C\}\ \mathcal{P}_a\ \{\mathcal{F}_2\}}{\{\mathcal{F}\}\ \texttt{if}\ C\ \texttt{then}\ \mathcal{P}_c\ \texttt{else}\ \mathcal{P}_a\ \texttt{endif}\ \{\mathcal{F}_1 \vee \mathcal{F}_2\}} \qquad \text{(P-IF)}$$

$$\frac{\{\mathcal{F} \wedge C\}\ \mathcal{P}\ \{\mathcal{F}\}}{\{\mathcal{F}\}\ \texttt{while}\ C\ \texttt{then}\ \mathcal{P}\ \texttt{endwhile}\ \{\mathcal{F} \wedge \neg C\}} \qquad \text{(P-WHILE)}$$

Figure 2.1: Classic Hoare Logic for Partial Correctness

$\{\mathcal{F}\}\ \mathcal{P}\ \{\mathcal{F}'\}\ t < z$ at the end of the `while` loop body. However, the loop invariant $\mathcal{F}$ implies that t is always positive, and therefore, the loop cannot run forever. The ability to infer termination makes it possible to prove *liveness properties*, which guarantee that a program point is definitely reached.

As far as sequential programs are concerned, most often the properties one would be interested in are related to total correctness. Most sequential programs are expected to terminate, and upon termination, to deliver a certain result that can be expressed as a safety property. However, recent embedded and reactive systems related works [HR98, AHH93, HTZ96] are dealing with the verification of sequential programs that operate in a concurrent context. Such programs access shared resources and are expected to follow a certain policy in order to operate correctly.

In such a context, it is usually sufficient to verify that each sequential program in the system complies with the given policy, without regard to the concurrent context in which the program executes. However, since such programs do not usually terminate, and since the policy is expressed in a trace-based manner, rather than a state based manner, the usual total correctness approach to verification is not very useful in this context. The concepts of safety, liveness and progress, typical of concurrent program verification, must now be brought into the realm of sequential program reasoning, which is the topic of this work.

When speaking about *liveness* the typical understanding is that, once a liveness property is specified, the correctness of that property entails that at least one instance of the behaviors or states satisfying the property occurs during program execution. In this work, we shall also deal with a different kind of liveness properties, which we call *explicit liveness*. The correctness of an explicit liveness property implies that *all* behaviors or states satisfying the property are encountered during program execution. An example of such a property is that, given a variable x in a program $\mathcal{P}$, the number of distinct values that this variable takes at a specific program point is a number k. Since the program may not terminate, proving such a property may not be done by proving some safety property, and then termination. Proving explicit liveness properties is, throughout this work, the vehicle we use to prove weaker liveness properties in a parameterized manner; such parametericity will ensure that the proofs we produce are generic and allow their use in a compositional framework.

In conclusion, program verification has the main advantage that the verifier or checker employed avoids fixpoint computations. So the constraints or equations corresponding to the verification conditions are not solved. This means that an inductive argument has to be provided, typically by the user. Since the implication

$$\frac{[\mathcal{F}_1]\ \mathcal{P}\ [\mathcal{F}_2]}{[\mathcal{F}'_1]\ \mathcal{P}\ [\mathcal{F}'_2]} \qquad \begin{array}{l}\mathcal{F}'_1 \to \mathcal{F}_1\\ \mathcal{F}_2 \to \mathcal{F}'_2\end{array} \qquad \text{(T-CONSEQ)}$$

$$\frac{}{[\mathcal{F}]\ \texttt{skip}\ [\mathcal{F}]} \qquad \text{(T-SKIP)}$$

$$\frac{}{[\mathcal{F}[E/x]]\ x := E\ [\mathcal{F}]} \qquad \text{(T-ASSIGN)}$$

$$\frac{[\mathcal{F}]\ \mathcal{P}_1\ [\mathcal{F}']\quad [\mathcal{F}']\ \mathcal{P}_2\ [\mathcal{F}'']}{[\mathcal{F}]\ \mathcal{P}_1\ ;\ \mathcal{P}_2\ [\mathcal{F}'']} \qquad \text{(T-SEQ)}$$

$$\frac{[\mathcal{F} \wedge C]\ \mathcal{P}_c\ [\mathcal{F}_1]\quad [\mathcal{F} \wedge \neg C]\ \mathcal{P}_a\ [\mathcal{F}_2]}{[\mathcal{F}]\ \texttt{if}\ C\ \texttt{then}\ \mathcal{P}_c\ \texttt{else}\ \mathcal{P}_a\ \texttt{endif}\ [\mathcal{F}_1 \vee \mathcal{F}_2]} \qquad \text{(T-IF)}$$

$$\frac{[\mathcal{F} \wedge C \wedge t = z]\ \mathcal{P}\ [\mathcal{F} \wedge t < z]}{[\mathcal{F}]\ \texttt{while}\ C\ \texttt{then}\ \mathcal{P}\ \texttt{endwhile}\ [\mathcal{F} \wedge \neg C]} \quad \mathcal{F} \to t \geq 0 \qquad \text{(T-WHILE)}$$

Figure 2.2: Hoare Logic for Total Correctness

involved in the verification condition is itself undecidable, the proof verification can only be partially automated, even though the solution to the equations or constraints is provided. Therefore interaction of the programmer with the prover is ultimately needed, and that is somewhat a shortcoming, in particular because the size of the proof is often exponential in the program size.

An alternative [LN98] consists in restricting the form of predicates considered by the prover. This can cause unsound verification condition simplifications, essentially to make the verifier simpler. Because theorem provers are driven by unformalized heuristics, and these heuristics and their interactions are changed over time for improving proof strategies, theorem provers are often unstable over time, in the sense that proof strategies get changed so that old proofs no longer work. Another

weakness which makes interaction with other formal methods somewhat difficult is the uniform encoding of properties as syntactical terms or formulas. It follows that the theorem prover has ultimately to be extended with program analyzers, model checkers, typing, among others [Sha96, SSJ+96], in particular for mechanizing and combining abstractions. We hope that the present work makes a contribution in that direction as well.

2.2 Abstract Interpretation

Since program verification deals with properties, or more precisely, sets of objects with these properties, abstract interpretation can be formulated in an application-independent setting, as a theory for approximating sets and set operations. A more restricted understanding of abstract interpretation is to view it as a theory of approximation of the behavior of a dynamic discrete system. Since such behaviors can be characterized by fixpoints [CC79a, Tar55], an essential part of the theory provides constructive and effective methods for fixpoint approximation and checking by abstraction [CC79b, CC92c].

The semantics of a programming language defines the semantics of any program written in this language. The semantics of a program provides a formal mathematical model of all possible behaviors of a computer system executing this program in interaction with any possible environment. We typically express (abstractions of) a computation by means of a monotonic mapping, typically called a semantic transformer, whose application represents (an abstraction of) a computation step. We shall first introduce the requisite concepts and results concerning monotonic mappings and their fixed points, as they are given in [Llo84].

A relation $\leq$ on a set S is a *partial order* if it is reflexive, transitive and antisym-

metric. If a partial order $\leq$ is defined on a set S, we say that $(S, \leq)$ is a *partially ordered set*, or a *poset*. We will also lift partial orders to mappings. Given two mappings $T_1 : S \mapsto S$ and $T_2 : S \mapsto S$, we say that $T_1 \leq T_2$ if $T_1(x) \leq T_2(x)$ for all $x \in S$.

If $(S, \leq)$ is a poset, we say that $a \in S$ is an *upper bound* of a set $X \subseteq S$ if $x \leq a$ for all $x \in X$. We also say that $b \in S$ is a *lower bound* of a set $X \subseteq S$ if $b \leq x$ for all $x \in X$. Moreover, we say that an element $M \in S$ is the *least upper bound* of a subset X of S if for all upper bounds a we have $M \leq a$. If such an element M exists, we denote it by $lub(X)$. Similarly, we say that an element $m \in S$ is the *greatest lower bound* of a subset X of S if for all lower bounds b we have $b \leq m$. If such an element m exists, we denote it by $glb(X)$.

A poset L is a *complete lattice* if $lub(X)$ and $glb(X)$ exist for every subset X of L. We let $\top$ denote the *top element* $lub(L)$, and $\bot$ denote the *bottom element* $glb(L)$. We also say that a subset X of a complete lattice L is *directed* if every finite subset of X has an upper bound in X.

If L is a complete lattice, a mapping $T : L \mapsto L$ is *monotonic* if $T(x) \leq T(y)$ whenever $x \leq y$. We say that T is *continuous* if $T(lub(X)) = lub(T(X))$, for every directed subset X of L. It is immediate to prove that a continuous mapping is also monotonic.

Let L be a complete lattice and $T : L \mapsto L$ be a mapping. We say that $a \in L$ is the *least fixed point* of T if a is a fixed point (that is, $T(a) = a$) and for all fixed points b of T, we have $a \leq b$. An element $c \in L$ is a *postfixpoint* of the operator T if $T(c) \leq c$.

2.1 Proposition Let L be a complete lattice and $T : L \mapsto L$ be a monotone mapping. Then, $lfp(T)$ exists and is equal to $glb(\{X \mid T(X) \subseteq X\})$

Proof: A proof can be found in [NNH99]. □

We also require the concept of ordinal powers of T. First we recall some elementary properties of ordinal numbers. The number 0 is represented by $\emptyset$. Then we define $1 = \{\emptyset\} = \{0\}$, $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\}$, and so on. The first infinite ordinal is $\omega = \{0, 1, 2, \cdots\}$, the set of all natural numbers. We will denote finite ordinals by the letters $n, m, \cdots$, while arbitrary ordinals will be denoted by the Greek letter β, possibly subscripted. We can specify an ordering $<$ on the collection of all ordinals by defining $n < m$ if $n \in m$. For example, $n < \omega$, for all finite ordinals. We will normally write $n \in \omega$ rather than $n < \omega$. If β is an ordinal, the *successor* of β is the ordinal $\beta + 1 = \beta \cup \{\beta\}$, which is the least ordinal greater than β. The ordinal $\beta + 1$ is then said to be a *successor ordinal*. If β is a successor ordinal, say $\beta = \beta' + 1$, we denote β' by $\beta - 1$. An ordinal β is a *limit ordinal* if it is not the successor of any ordinal. The smallest limit ordinal, apart from 0, is ω.

We will also require the principle of transfinite induction, which is as follows. Let $\mathcal{P}(\beta)$ be a property of ordinals. Assume that for all ordinals β_0, if $\mathcal{P}(\beta_1)$ holds for all $\beta_1 < \beta_0$, then $\mathcal{P}(\beta_0)$ holds. Then $\mathcal{P}(\beta)$ holds for all ordinals β.

We now define the ordinal powers of T.

2.2 Definition Let L be a complete lattice and $T : L \mapsto L$ a mapping. Then we define:

$T \uparrow 0 = \bot$.

$T \uparrow \beta = T(T \uparrow (\beta - 1))$, if β is a successor ordinal.

$T \uparrow \beta = lub(\{T \uparrow \beta' \mid \beta' < \beta\})$, if β is a limit ordinal.

□

2.3 Proposition Let L be a complete lattice and $T : L \mapsto L$ be continuous. Then $lfp(T) = T \uparrow \omega$.

Proof: A proof can be found in [Llo84]. □

Let S be a set, $T : 2^S \mapsto 2^S$ a continuous mapping defined on the complete lattice $(2^S, \subseteq)$, and A a subset of 2^S. We denote by $T \cup A$ the mapping $\lambda X \,.\, T(X) \cup A$.

2.4 Proposition Consider a set S, a subset A of S, and a continuous mapping $T : 2^S \mapsto 2^S$. The mapping $T \cup A$ is continuous.

Proof: It is easy to prove that the composition of two continuous mappings is continuous, and that for any $A \subseteq S$, the mapping $\lambda X \,.\, X \cup A$ is continuous. Therefore, if T is continuous, so is $T \cup A$. □

It is often very convenient to express the semantics of a programming language as the fixed point of some operator. Program reasoning techniques either compute an element of the lattice that is larger than the semantics (analysis), or attempt to prove that a given element is larger than the semantics (verification). Proposition 2.1 shows that post-fixpoints are larger than the least fixed point for a continuous operator, and therefore it is often convenient to prove that a desired approximation of the semantics is a postfixpoint.

2.5 Remark Given a complete lattice L, a continuous operator $T : L \mapsto L$ and a set $A \subseteq L$, it can be easily proved that $(T \cup A) \uparrow n = T^n(A)$. It also follows immediately that $lfp(T \cup A) = T^\omega(A)$, and that $lfp(T \cup A)$ is a fixpoint of T. □

2.6 Proposition Let L be a complete lattice and T_1, T_2 two continuous mappings on L such that $T_1 \subseteq T_2$. Then, any post-fixpoint of T_2 is a post-fixpoint of T_1.

Proof: Let X be a postfixpoint of T_2. Then, $T_2(X) \subseteq X$. Also, from the hypothesis we have $T_1(X) \subseteq T_2(X)$. The transitivity of the $\subseteq$ relation entails $T_1(X) \subseteq X$, which proves that X is a postfixpoint of T_1. $\square$

The following remark shall be useful in Section 14.3, where we discuss propagation for program reasoning.

2.7 Remark Let L be a complete lattice, and $T : L \mapsto L$ a monotone operator. Then, any element $X \in L$ is a postfixpoint of the operator $T \cap I$. $\square$

Next we discuss Galois connections. A Galois connection is a particular correspondence between two partially ordered sets, originally invented by Galois [Bir40] for the study of symmetry groups. In program reasoning, whose object of study could be summarized as "approximating program properties", Galois connections are useful in designing sets of approximations, or *abstract domains*, and providing an interpretation to every approximation. The theory of *abstract interpretation* [CC77a, CC92c] provides a framework for the design of program analyzers whose abstract domain is Galois connected with the concrete domain of discourse. Cousot [Cou02] has also shown that all semantics can be hierarchized using a Galois-connection-based partial order.

2.8 Definition Two complete lattices $(L, \subseteq_L)$ and $(M, \subseteq_M)$ are *Galois connected* iff there exist two monotone mappings $\alpha : L \mapsto M$ and $\gamma : M \mapsto L$ such that $\lambda x \,.\, x \subseteq_L \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq_M \lambda y \,.\, y$. $\square$

We call α the *abstraction* function, and γ the *concretization* function. We represent the fact that the lattices $(L, \subseteq_L)$ and $(M, \subseteq_M)$ are Galois connected via the mappings α and γ by $(L, \subseteq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq_M)$.

2.9 Definition A Galois connection $(L, \subseteq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq_M)$ is a *Galois insertion* if $\lambda x \,.\, x \subseteq_L \gamma \circ \alpha$ and $\alpha \circ \gamma = \lambda y \,.\, y$. $\square$

2.10 Proposition Let $(L, \subseteq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq_M)$ be a Galois connection, and $f : L \mapsto L$ and $g : M \mapsto M$ two monotone mappings such that $\alpha \circ f \circ \gamma \subseteq_M g$. Then, $\mathit{lfp}(f) \subseteq_L \gamma(\mathit{lfp}(g))$ and $\alpha(\mathit{lfp}(f)) \subseteq_M \mathit{lfp}(g)$.

Proof: A proof can be found in [NNH99]. $\square$

Given two lattices $(L, \subseteq_L)$ and $(M, \subseteq_M)$, and a mapping $\alpha : L \mapsto M$, we say that α is *strict* if $\alpha(\bot_L) = \bot_M$. The following proposition states the Kleenian fixpoint transfer theorem.

2.11 Proposition Let $(L, \subseteq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq_M)$ be a Galois connection, such that α is strict, and consider two continuous mappings $f : L \mapsto M$ and $g : M \mapsto L$ that satisfy the commutation condition $g \circ \alpha = \alpha \circ f$. Then, $\mathit{lfp}_M(g) = \alpha(\mathit{lfp}_L(f)$.

Proof: A proof can be found in [Cou02]. $\square$

A *fixpoint semantics specification* is a pair (L, T), where L is a complete lattice called the *semantic domain*, and $T : L \mapsto L$ is a continuous mapping called the *semantic transformer*. The semantic transformer captures (an abstraction of) a single computation step of that machine or automaton. Therefore, we shall be able to express any of the semantics of a programming language as the least fixpoint of some semantic transformer T. This property is very important, because by further abstracting the semantic transformer we can create means of analyzing or verifying programs in finite time.

A programming language semantics is more or less precise according to the considered observation level of program execution [AP81]. This intuitive idea can be formalized by abstract interpretation [Cou02] and applied to different languages [Gia96], including for proof methods [Cou90, CGLV03]. The theory of abstract interpretation formalizes this notion of approximation and abstraction in a mathematical setting which is independent of particular applications. In particular, abstractions must be provided for all mathematical constructions used in semantic definitions of programming and specification languages [CC79b, Mar90, Nie82, RK92, Sch95, CC92c].

An abstract domain is an abstraction of the concrete semantics in the form of abstract properties and abstract operations. Abstract domains for complex approximations are designed by composing abstract domains for simpler components [CC79b]. If the approximation is coarse enough, the abstraction of a concrete semantics can lead to an abstract semantics which is less precise, but is effectively computable. By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them [CC77c]. Abstract interpretation algorithms provide approximate methods for computing this abstract semantics. The most important algorithms in abstract interpretation are those providing effective methods for the exact or approximate iterative resolution of fixpoint equations [CC77a].

The abstraction idea and its formalization are equally applicable in other areas of computer science such as proof checking [GV96], automated deduction, theorem proving [FT89], etc.

There is a wealth of program semantics that have been used to express various aspects of the behavior of the program during its execution. Our finer grain of observation of program execution, that is the most precise of the semantics that

we will consider, is that of a trace semantics [Cou01, Cou02]. An execution of a program for a given specific interaction with its environment is a sequence of states, observed at discrete intervals of time, starting from an initial state, then moving from one state to the next state by executing an atomic program step or transition and either ending in a final regular or erroneous state or non-terminating, in which case the trace is infinite. The relational semantics [HL74a, MT91] can be obtained from the trace semantics by replacing every trace by a pair of the initial and final state of the trace. For infinite trace, the final state is denoted by $\perp$. The denotational semantics [Ten76] can be obtained from the relational semantics by defining a partial mapping that returns the final state for every initial state. Of course, the mapping is partial since a program does not necessarily terminate for every initial state. The abstraction from relational to big-step operational, or natural semantics simply consists in forgetting everything about non-termination. The small-step operational, or transition semantics consists of collecting the set of all pairs of states that appear consecutively in a trace. A further abstraction consists in collecting all states appearing along some finite or infinite trace. This is the partial correctness semantics [Nau66, Cou90, Flo67, Hoa69, HW72] or the static/collecting semantics [CC77a] for proving invariance properties of programs.

There is a wide range of effective abstractions used in practice. We mention the following categories: non-relational, relational, and symbolic abstractions. The non-relational, attribute-independent, or Cartesian abstractions [CC79b] consist in ignoring the possible relationships between the values of program variables. It follows that a set of pairs is approximated through projection by a pair of sets. Each such set may still be infinite, and in general, not exactly computer representable, and therefore, further abstractions may be needed. The sign abstraction [CC79b] consists in replacing integers by their signs, thus ignoring their absolute value. The

interval abstraction [CC77c, CC77a] is more precise, since it approximates a set of integers by its minimal and maximal values. The congruence abstraction [Gra97] maps an integer into its remainder modulo n.

Relational abstractions are more precise than non-relational ones [JM80] in that some of the relationships between values of the program states are preserved by the abstraction. For example, the polyhedral abstraction [CH78] approximates a set of tuples of integers by their convex hull.

Symbolic abstractions approximate the symbolic structures manipulated by programs. Among such structures, we mention control structures (control graphs), data structures (search trees, pointers) [Ven99], communication structures [Ven98, Fer00, HJNN99], etc.

A compromise between semantic expressibility and algorithmic efficiency was recently introduced by [Mau00], using Binary Decision Graphs and Tree Schemata to abstract infinite sets of infinite trees.

Chapter 3

Related Work

The work in this thesis has been influenced by concepts, techniques, algorithms, and calculi developed in the areas of program analysis, model checking, verification condition generators, proof carrying code, and higher order logic. In this chapter we review related work that had a more or less direct impact on the development of this research.

3.1 Program Analysis

Program analysis is a termed coined to represent a wealth of techniques for predicting computable approximations to the set of values or behaviors arising dynamically at run-time during the execution of a program. The initial application of program analysis was to allow compilers to generate optimized code, for example, by avoiding redundant computations, or reusing available results, or performing constant propagation, that is, replacing expressions by their value if that value is constant and known at compile time. Among the more recent applications are various kinds of code certification, for example, to detect bugs, or to reduce the likelihood of malicious, or unintended behavior.

The literature on program analysis is quite extensive. In this section, we shall provide a short review of various approaches, without aiming to be exhaustive. In the previous chapter we have given a more detailed overview of abstract interpretation, which is one of the multitude of program analysis approaches.

There are two main themes behind all approaches to program analysis. The first one is that in order to remain computable, one can only provide *approximate answers*. The other one is that all program analyzes are semantics-based; this means that the information obtained from the analysis can be proved to be correct with respect to a semantics of the programming language. In practice, the program analyzer contains a generator reading the program text and producing equations or constraints whose solution is a computer representation of the program abstract semantics. A solver is then used to produce solutions to these abstract equations or constraints. A popular resolution method is to use iteration. If the iteration doesn't reach a fixpoint in a finite number of steps, or if the number of steps is too large to be afforded, the convergence may have to be ensured or accelerated using a widening operator that overestimates the solution in finitely many steps, followed by the application of a narrowing operator, to improve the approximation [CC77a, CC92a]. In abstract compilation, the generator and solver are directly compiled into a program which directly yields the approximate solution [MRB98]. This solution is an approximation of the abstract semantics which is then used by a diagnoser to check the specification. Because of the loss of information, the answer given by the diagnosis process is on of yes, no, unknown or irrelevant. Besides diagnosis, program analysis is also used for other applications in which case the diagnoser may be replaced by an optimizer, in the case of compile-time optimization, a program transformer, in the case of partial evaluation [Jon97], and so on.

The main advantage of program analysis is that it is completely automatic: It

has been applied to pieces of code comprising more than one million lines without user interaction. The abstractions are chosen to be of wide scope without specialization to a particular program. Abstract domains can be designed and implemented into libraries which are reusable for different programming languages. The objective is to discover invariants that are likely to appear in many programs so that the abstraction would be widely reusable and the program analyzer be of economic interest. The drawback of this general scope is that the considered abstract specifications and properties are often simple, mainly concerning elementary safety properties such as absence of run-time errors. For example non-linear abstractions of sets of points are very difficult and very few mathematical results are of practical interest and directly applicable to program analysis [YcFG+00]. Checking termination and similar liveness properties is trivial with finite state systems, at least from a theoretical if not algorithmic point of view (e.g. finding loops in finite graphs). The same problem is much more difficult for infinite state systems because of fairness or of potentially infinite data structures, as considered in partial evaluation [JGS93]. The existence of data structures of unbounded size may lead to infinite cycles and hence termination or inevitability proofs require the discovery of variant functions on well-founded sets, which is very difficult in full generality [CC94]. Even when considering restricted simple abstract properties, the semantics of real-life programming languages is very complex, due to aspects like recursion, concurrency, modularity, etc., and as a result, so is the corresponding abstract interpreter. The abstraction of such a semantics, and therefore the design of the analyzer is mostly manual, and typically beyond the ability of casual programmers or theorem provers. The considered abstractions must have a large scope of application and must be easily reusable to be of economic interest. From a user point of view, the results of the analysis have to be presented in a simple way, for example by

pointing at errors only or by providing abstract counter-examples, or less frequently concrete ones.

There are four main approaches to program analysis: data flow analysis [CC79b, CC00b, Sch98a]; constraint-based analysis [CC95, Aik99, FA97, Hei92a, Hei92b, Hei94, Hei95], type inference [Cou97], and abstract interpretation [CC77a]. It has been applied to procedural languages [Bou90, Bou92], object-oriented languages [HT98a], functional languages [Hen91, HS95, NS95, NN88, Pal93] logic engages including Prolog [CC92b, Deb94, BGL93, BCM95, CGS89, CDG93a], constraint logic languages [GDL93, Han93, Han95, MS94], and concurrent constraint logic languages [CFMW93].

There is a very extensive range of applications of program analysis, among which we mention program optimization [CGS94, CS92] and transformation [Nie85], alias analysis [DMW98], abstract debugging, testing and verification [CGLV99, Cou81, CC00a, FFK$^+$96], cache and pipeline behavior prediction [FMWA99], dependency analysis [Blu99], path/trace analysis [CL96], closure analysis [Pal95], control flow analysis [DP97], compile-time garbage collection [Hug92], binding time analysis [Hen91, HS95, NN88], strictness analysis [CC93b, Con94, DW90, Hen94, HY86, Myc80, Nie87, NN93], occurs check reduction [Søn86], groundness analysis [MS93], sharing analysis [CS98].

3.2 Model Checking

Model checking [CE81, MOSS99] has been very successful for the verification of hardware [BCRZ99], communication protocols [CJM00, BG99], and real-time [HNSY92] [CCMM95] processes. As far as software systems are concerned, the question for the next decade is whether model checking can be extended to the veri-

fication of very large real-life programs. The idea behind software model-checking is that first a model of the program must be designed, typically as a manually designed abstraction of the program semantics, which could be in the form of a transition system similar to a small step operational semantics. Then a specification of the program must be provided by the user in a very expressive temporal logic [BAMP83], A model checker can then check the specification by exhaustive search or symbolic exploration of the state space. The spectacular success of model checking followed from the clever design of data structures as BDDs [Ake78] or QDDs [BGWW97], and algorithms as minimal state graph generation [BFH91], fixpoint computation or SAT [BCCZ99, BCC$^+$99], for representing very large sets of booleans and their transformations. The approximation is that the model must be finite-state or some form of abstraction must be used [PDD97] to reduce the verification problem to finite state. Another trend in finite-state model checking is to consider safety properties only and polyhedra abstractions, with variants (e.g.Presburger arithmetic [GBP97]). This is a direct application of polyhedra static program analysis [CH78], including the use of widenings.

Although model checking gained a factor of 100 in 10 years, it is very difficult to scale up because of the state explosion problem. So, the necessary restriction to available computer resources often reduces the model checker from formal verification to debugging on part of the state space. Since the model must ultimately be finite to allow for exhaustive search/symbolic exploration, abstraction is mandatory, which is a very difficult task to do manually. Moreover, some forms of abstractions — such as interval [CC77a] or polyhedra [CH78] abstractions — do not abstract concrete transition systems into abstract transition systems so that the model checker may not be reusable in the abstract [CC99]. It follows that abstractions are difficult and not reusable, hence not cost effective.

3.3 Verification Condition Generators

Given a Hoare logic for a particular programming language, it is possible to partially automate the process of applying the rules of the logic to prove the correctness of a program. Generally this process is guided by the structure of the program, applying in each case the Hoare logic rule for the command which is the major structure of the phrase under consideration. A *verification condition generator* [ILL73, Rag73, Gra87, Gor88, Age92, Mel94, CM92, CRR$^+$93, HM98, HM94] takes a suitably annotated program and its specification, and traces a proof of its correctness, according to the rules of the language's axiomatic semantics. Each command has its own appropriate rule which is applied when that command is the major structure of the current proof goal. This replaces the current goal by the antecedents of the Hoare rule. These antecedents then become the subgoals to be resolved by further applications of the rules of the logic. Verification condition generators were hailed, in their very beginning, as an answer to the difficulty of proving programs correct. This hope waned over time, however. First of all, it was discovered that for many simple programming languages, the work done by the verification condition generator was mostly trivial and not hard to do by hand. Then, even after the verification condition generator had done its work and reduced the problem of proving the program to the problem of proving the verification conditions, those verification conditions were not always easy to prove, and could contain the bulk of the necessary effort of the entire proof. An additional feature that was not discussed as much was the fact that for the most part, these verification condition generators were not themselves verified. This meant that any proof using and relying on these tools might not be sound, even if all the verification conditions were correctly proved. [Rag73] is a notable exception to this, far ahead of its time. Finally, a verification

condition generator is usually based on an axiomatic semantics for the programming language. When these programming languages were extended to include procedure calls (an obvious necessity), a disturbing number of the rules proposed for procedure calls turned out to be unsound. It became evident that the area of procedure calls was more complicated than had originally appeared. In recent years, there have been several shallow embeddings of programming languages in the theorem proving environment, including the creation of verification condition generators. These have taken the form of tactics, which in general reduce a current goal to be proved to a sufficient set of subgoals. In contrast to the traditional verification condition generators created as stand-alone programs, these verification condition generators had their soundness secured by the inherent security of the system itself. This was a very significant advantage. No verification of the verification condition generator itself was necessary, as every application of the tactic would prove all necessary subsidiary theorems as part of the process. However, this also was a weakness of the verification condition generator, because it required that every proof be carried out at the semantic level, instead of the syntactic manipulations that were simpler and that were the traditional work of verification condition generators. Also, these semantic verification condition generators required an additional degree of annotation and specification from the user beyond what had been required by the syntactic verification condition generator.

3.4 Extended Static Checking

The extended static checking system [Lei01, FLL+02] is a checker aimed at statically detecting simple errors in programs written in Modula-3 and Java: null dereferences, out-of-bounds array indices, or simple deadlocks or race conditions in concurrent

programs. The checker attempts to achieve these goals using a quite general program verification framework. The user annotates the program being checked with specifications, and a verification condition generator transforms the program and specification into a logical formula whose validity ensures the absence of the errors being considered. This formula is passed to the automatic theorem prover "Simplify," developed expressly for the extended static checker. If the prover is unable to prove that the errors do not occur, it returns an assignment of values to program variables that falsifies the formula. This information can be presented to the programmer, giving information about the error. The aim of the authors of this system is that this kind of limited verification will be seen in the future much as type-checking is viewed today.

The specification language of the checking system allows several kinds of invariants. The methodology for generating verification conditions requires the programmer to annotate most loops with loop invariants. Users may enter assertions that a predicate must hold at a particular point in a program. The programmer may also direct the checker to assume without proof that a predicate holds at some point. Finally, an invariant expresses a predicate about some state variables that must be preserved by all procedures that modify those variables. The more novel aspects of the specification language deal with abstraction. Modula-3 and Java are modular languages; their interfaces are supposed to hide implementation details, allowing a range of implementation choices and protecting clients from implementation changes. The interface allows the declaration of abstract variables that represent the manipulated state, and the specification of procedures in terms of these abstract variables. The implementation specification will give an abstraction function to connect the abstract variables with the concrete implementation.

3.5 Proof Carrying Code

Proof-carrying code [Nec97a] is a technique for safe execution of untrusted code. The basic idea is to require the code producer to generate a formal proof that the code meets the safety requirements set by the code receiver. The code receiver can easily check the proof by using a simple and easy-to-trust proof checker.

PCC has many uses in systems whose trusted computing base is dynamic, either because of mobile code or because or regular bug fixes or updates. Examples include, but are not limited, to extensible operating systems, Internet browsers capable of downloading code, active network nodes and safety-critical embedded controllers.

3.6 Higher Order Logic

The family description and assertion languages we use in the later chapters of this thesis have been largely inspired by Higher Order Logic [GM93]. Higher Order Logic (HOL) is a mechanical proof assistant that mechanizes higher order logic, and provides an environment for defining systems and proving statements about them. It is secure in that only true theorems may be proved, and this security is ensured at each point that a theorem is constructed. HOL has been applied in many areas. The first and still most prevalent use is in the area of hardware verification, where it has been used to verify the correctness of several microprocessors. In the area of software, has been applied to Lamport's Temporal Logic of Actions (TLA), Chandy and Misra's UNITY language, Hoare's CSP, and Milner's CCS and π-calculus. HOL is one of the oldest and most mature mechanical proof assistants available, roughly comparable in maturity and degree of use with the Boyer-Moore Theorem Prover [BM90]. Many other proof assistants have been introduced more recently that in some ways surpass HOL, but HOL has one of the largest user communities and

history of experience.

Higher Order Logic is a version of predicate calculus which allows quantification over predicate and function symbols of any order. It is therefore an ω-order logic, according to Andrews [And86]. In such a type theory, all variables are given types, and quantification is over the values of a type. Type theory differs from set theory in that functions, not sets, are taken as the most elementary objects. Some researchers have commented that type theory seems to more closely and naturally parallel the computations of a program than set theory. A formulation of type theory was presented by Church in [Chu40]. Andrews presents a modern version in [And86] which he names $\mathcal{Q}_0$. The logic implemented in the Higher Order Logic system is very close to Andrews' $\mathcal{Q}_0$. This logic has the power of classical logic, with an intuitionistic style. The logic has the ability to be extended by several means, including the definition of new types and type constructors, the definition of new constants (including new functions and predicates), and even the assertion of new axioms.

The HOL logic is based on eight rules of inference and five axioms. These are the core of the logical system. Each rule is sound, so one can only derive true results from applying them to true theorems. As the system is built up, each new inference rule consists of calls to previously defined inference rules, ultimately devolving to sequences of these eight primitive inference rules. Therefore the proof system is fundamentally sound, in that only true results can be proved. HOL provides the ability to assert new axioms; this is done at the user's discretion, and he then bears any responsibility for possible inconsistencies which may be introduced.

As a mechanical proof assistant, the HOL system provides the user a logic that can easily be extended, by the definition of new functions, relations, and types. These extensions are organized into units called *theories*. Each theory is similar to

a traditional theory of logic, in that it contains definitions of new types and constants, and theorems which follow from the definitions. It differs from a traditional theory in that a traditional theory is considered to contain the infinite set of all possible theorems which could be proved from the definitions , whereas a theory in HOL contains only the subset which have been actually proved using the given rules of inference and other tools of the system. When the system it started, it presents to the user an interactive programming environment using the programming language ML. The user types expressions in ML, which are then executed by the system, performing any side effects and printing the value yielded. The language contains the data types *term* and *thm*, which represent terms and theorems in the logic. These terms represent a second language, called the *object language*. ML functions are provided to construct and deconstruct terms of the language. Theorems, however, may not be so freely manipulated. Of central importance is the fact that theorems, objects of type *thm*, can only be constructed by means of the eight standard rules of inference. Each rule is represented as a function. Thus the security of HOL is maintained by implementing *thm* as an abstract data type. Additional rules, called *derived rules of inference*, can be written as new functions. A derived rule of inference could involve thousands of individual calls to the eight standard rules. Each rule typically takes a number of theorems as arguments and produces a theorem as a result. This methodology of producing new theorems by calling functions is called *forward proof*. One of the strengths of is that in addition to supporting forward proof, it also supports *backwards proof*, where one establishes a goal to be proved, and then breaks that goal into a number of subgoals, each of which is refined further, until every subgoal is resolved, at which point the original goal is established as a theorem. At each refinement step, the operation that is applied is called in HOL a *tactic*, which is a function of a particular type. The effect of applying a tactic is to

replace a current goal with a set of subgoals which, if proved, are sufficient to prove the original goal. The effect of a tactic is essentially the inversion of an inference rule. Tactics may be composed by functions called *tacticals*, allowing a complex tactic to be built to prove a particular theorem. Functions in are provided to create new types, make new definitions, prove new theorems, and store the results into theories on disk. These may then be used to support further extensions. In this incremental way a large system may be constructed.

3.7 Relevance to Our Work

An early version of this work appeared in [HJV00]. When building our framework, we have focused on accommodating four main directions of research in program reasoning: Hoare-style program verification, automated program reasoning methods, reasoning about behavior, and (assertion) languages to express program properties. In bringing in concepts related to these directions into our framework, our main concern was to keep the framework compositional and to make the reasoning program point-based. To us, the ESC and verification condition generators are realizations of Hoare style reasoning, program analysis methods are automated tools to perform program reasoning, and model checking is the realization of reasoning about behavior. We have been greatly influenced by ESC and verification condition generators in building our propagation-based reasoning, which, in its abstracted form can also realize program analysis methods.

Model checking has also contributed two useful ideas to our framework, namely, reasoning about behavior, and the ability to reason about finite-state programs by iterative application of a transfer function that reaches a fixpoint in finite time. As it shall be shown in Chapter 9 the progressive transfer function that we define has

a unique fixpoint, and computes the projection of a program's behavior for every program point. When applied to assertions, the propagation operator would refine them, and possibly prove them, as stated by Theorem 15.6. However, in the case of finite state systems, the fixpoint of the propagation operator would be computed simultaneously with refining the assertions, providing *exact* information about the program.

The Higher Order Logic framework has influenced the description and assertion languages that we use throughout this thesis.

In the next chapter, we start the concrete development of our framework by introducing a simple imperative language and a structural induction principle.

Chapter 4

Syntax

We introduce a simple imperative language over integers and arrays of integers, with assignment, conditional and iteration statements. This language allows the use of program point annotations as means of gathering information about the program in the process of verifying, analyzing or reasoning about it. We then define a small-step operational semantics for our language, and continue on with the definitions of trace semantics and a collecting semantics, and we show that these semantics can be represented as annotated programs. The contribution of this chapter is the *progressive semantics*, which is also defined as an annotated program. We show that the progressive semantics is hierarchically between the trace semantics and the collecting semantics by defining appropriate Galois connections. We also argue that the progressive semantics induces a more informative kind of reasoning, which supports verifying or reasoning about liveness and safety properties within the same framework.

4.1 A Simple Programming Language

We consider a simple imperative language over integers and arrays of integers, with assignment, conditional and iteration statements. The definition of the language allows for annotations that can be added at each program point. There are several kinds of annotations that would be of interest. On one hand, we need to uniquely identify each program point, and for this purpose we will use annotations from a set of labels, in such a way that each label appears only once in the annotated program. Apart from labels, it would be useful to attach assertions and program properties to every program point. Such assertions and program properties would be expressed as formulas, sets, or various kinds of mappings. Therefore, in general, an annotation would be a pair between a label and either a set, a formula, or a mapping.

The syntax of an annotated program is given in Figure 4.1. The definition is parameterized by a set of annotations **Annot**, whose specialization will produce various structures of interest. An *annotated program* is either a single annotation, or an *annotated statement* placed between two annotations[1]. The basic annotated statements are the null command `skip` and the assignment $x := E$. Two annotated statements $Stmt_1$ and $Stmt_2$ can be composed sequentially via the construct $\text{Stmt}_1\ \langle \mathcal{A} \rangle\ \text{Stmt}_2$, where $\mathcal{A}$ is an annotation, while two programs P_1 and P_2 can be composed conditionally, via the construct `if` C `then` P_1 `else` P_2 `endif`. An annotated program P can also be iterated sequentially via the construct `while` C `do` P `endwhile`, while C holds. In Figure 4.1 **AProg** denotes the non-terminal that generates annotated programs, **AStmt** denotes the non-terminal that generates annotated statements, **Var** denotes the non-terminal that generates the

[1]Having a single annotation as an annotated program is a notational convenience that would simplify certain definitions and proofs.

language of variables and **Expr** represents the non-terminal that generates the language of program expressions. Program variables can be either scalar, or array variables. Scalar variables are generated by the **SVar** non-terminal, whereas array variables are generated by the **AVar** non-terminal. Similarly, expressions can be either scalar expressions, generated by **SExpr**, or array expressions, generated by **AExpr**. Scalar expressions are the usual ones, either integer constants, or scalar expressions connected via the usual arithmetic operators, or an array element. Array expressions are of the form $a[E_1 \mapsto E_2]$, where a is an array variable, E_1 is a scalar expression that denotes the subscript, and E_2 is a scalar expression that denotes the new value of the array element $a[E_1]$. The value of such an expression is an array that is identical to a, with the exception of the element subscripted by E_1, whose value becomes E_2. The syntax of array expressions is useful in describing the assignment of array elements, and is different from the usual one, which is $a[E_1] = E_2$. We prefer this syntax since it allows for a uniform treatment of scalar and array assignment. Finally, **Constr** is the non-terminal that generates the language of constraints (boolean conditions). By abuse of language, we will also use the non-terminal symbols **AProg**(**Annot**), **Var**, **Expr**, and **Constr** to refer to the languages (sets) that these non-terminals generate. Moreover, we shall denote annotated programs by the letter P, expressions by E and constraints by C. We will denote variables by the letters x, y, z, possibly subscripted. Whenever we want to distinguish between scalar variables and array variables, we shall denote array variables by a, b, c.

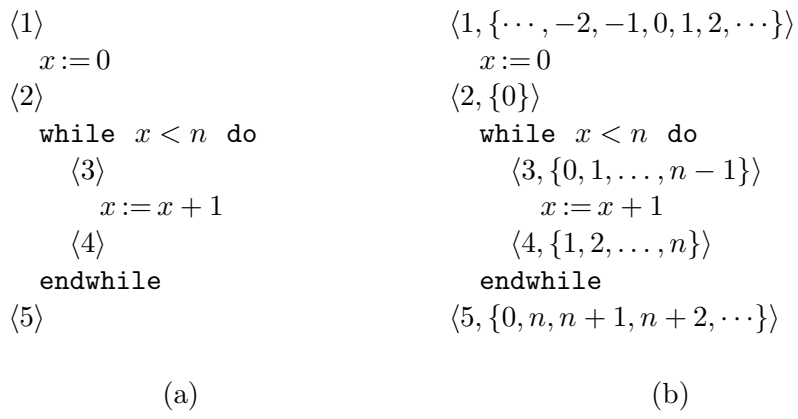
We note that the syntax of the language defined in Figure 4.1 may allow assignments that are not well-typed, in the form of a scalar variable being assigned an array expression, or an array variable being assigned a scalar expression. We could easily correct this problem at the expense of introducing two rules for assignment,

$$
\begin{array}{lcl}
\mathbf{AStmt(Annot)} & ::= & \mathbf{Var} := \mathbf{Expr} \\
& | & \texttt{skip} \\
& | & \texttt{if } \mathbf{Constr} \texttt{ then } \mathbf{AProg}_1 \\
& & \qquad \texttt{else } \mathbf{AProg}_2 \texttt{ endif} \\
& | & \texttt{while } \mathbf{Constr} \texttt{ do } \mathbf{AProg}_1 \texttt{ endwhile} \\
& | & \mathbf{AProg}_1 \ \langle \mathbf{Annot} \rangle \ \mathbf{AProg}_2 \\
\\
\mathbf{AProg(Annot)} & ::= & \langle \mathbf{Annot} \rangle \mid \langle \mathbf{Annot}_1 \rangle \ \mathbf{AStmt} \ \langle \mathbf{Annot}_2 \rangle \\
\\
\mathbf{Var} & ::= & \mathbf{SVar} \mid \mathbf{AVar} \\
\mathbf{SVar} & ::= & x \mid y \mid z \ \ldots \\
\mathbf{AVar} & ::= & a \mid b \ \ldots \\
\\
\mathbf{Expr} & ::= & \mathbf{SExpr} \mid \mathbf{AExpr} \\
\mathbf{SExpr} & ::= & \mathbf{SVar} \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots \\
& | & \mathbf{SExpr}_1 + \mathbf{SExpr}_2 \ \mid \ \mathbf{SExpr}_1 - \mathbf{SExpr}_2 \\
& | & \mathbf{SExpr}_1 * \mathbf{SExpr}_2 \ \mid \ \mathbf{SExpr}_1 / \mathbf{SExpr}_2 \\
& | & \mathbf{SExpr}_1 \ \% \ \mathbf{SExpr}_2 \ \mid \ + \mathbf{SExpr}_1 \mid \ - \mathbf{SExpr}_1 \\
& | & \mathbf{AVar} \ [\mathbf{SExpr}_1 \ ] \\
\mathbf{AExpr} & ::= & \mathbf{AVar} \mid \mathbf{AExpr}_1 [\mathbf{SExpr}_1 \mapsto \mathbf{SExpr}_2] \\
\\
\mathbf{Constr} & ::= & \mathbf{SExpr}_1 < \mathbf{SExpr}_2 \mid \mathbf{SExpr}_1 \leq \mathbf{SExpr}_2 \\
& | & \mathbf{SExpr}_1 > \mathbf{SExpr}_2 \mid \mathbf{SExpr}_1 \geq \mathbf{SExpr}_2 \\
& | & \mathbf{SExpr}_1 = \mathbf{SExpr}_2 \mid \mathbf{SExpr}_1 \neq \mathbf{SExpr}_2 \\
& | & \neg \mathbf{Constr}_1 \\
& | & \mathbf{Constr}_1 \wedge \mathbf{Constr}_2 \mid \mathbf{Constr}_1 \vee \mathbf{Constr}_2 \\
& | & \mathbf{Constr}_1 \Rightarrow \mathbf{Constr}_2
\end{array}
$$

Figure 4.1: The Syntax of Annotated Programs

one for scalars, and the other for arrays. However, having two assignment rules will make many of the proofs in this thesis more complicated. For this reason, we shall keep the current definition and assume in what follows that all assignments are well-typed.

Figure 4.2 presents two examples of annotated programs. In program (a), the annotations are simply labels (in this case natural numbers) that would allow us to uniquely identify the program points. We call such a program a *labeled program*. In example (b) we have as annotations pairs of labels and sets of values. The set



(a) (b)

Figure 4.2: Example of Annotated Programs

of integers associated with each label represents the values of variable x at each program point during the execution of the program. Such an annotated program, whose annotations are pairs of labels and sets, or pairs of labels and mappings, is called a *configuration*. We shall define several kinds of configurations later in the thesis.

Annotated programs are the central structure of this thesis. They are versatile enough to represent programs as well as to support the process of analyzing, verifying or reasoning about a program. An important attribute that adds to this versatility is the ability to reason inductively over annotated programs. We note however that the definition of the annotated programming language, while rigorous, leads to an unnecessarily complicated induction principle due to the mutually recursive nonterminals **AProg** and **AStmt**. The induction principle can be greatly simplified by defining the concatenation of two annotated programs $P_1 = \langle A_{1s} \rangle\, \mathit{Stmt}_1\, \langle A_{1f} \rangle$ and $P_2 = \langle A_{2s} \rangle\, \mathit{Stmt}_2\, \langle A_{2f} \rangle$ to be the program $P_1 \fatsemi P_2 = \langle A_{1s} \rangle\, \mathit{Stmt}_1\, \langle A_{1f} \rangle\, \mathit{Stmt}_2\, \langle A_{2f} \rangle$. The equality sign used above stands for syntactic identity.

4.1 Remark We note that exactly one of the following statements is true of an annotated program P.

a) There exist an annotation A such that $P = \langle A \rangle$.

b) There exist unique $A_s, A_f \in \mathbf{Annot}$ such that $P = \langle A_s \rangle \, \texttt{skip} \, \langle A_f \rangle$.

c) There exist unique $A_s, A_f \in \mathbf{Annot}$, $x \in \mathbf{Var}$ and $E \in \mathbf{Expr}$ such that $P = \langle A_s \rangle \, x := E \, \langle A_f \rangle$.

d) There exist unique $A_s, A_f \in \mathbf{Annot}$, $C \in \mathbf{Constr}$ and $P_1, P_2 \in \mathbf{AProg}(\mathbf{Annot})$ such that $P = \langle A_s \rangle \, \texttt{if} \; C \; \texttt{then} \; P_1 \; \texttt{else} \; P_2 \; \texttt{endif} \, \langle A_f \rangle$.

e) There exist unique $A_s, A_f \in \mathbf{Annot}$, $C \in \mathbf{Constr}$ and $P' \in \mathbf{AProg}(\mathbf{Annot})$ such that $P = \langle A_s \rangle \, \texttt{while} \; C \; \texttt{do} \; P' \; \texttt{endwhile} \langle A_f \rangle$.

f) There exist unique $P_1, P_2 \in \mathbf{AProg}(\mathbf{Annot})$ with the last annotation of P_1 being the same as the first annotation of P_2 such that $P = P_1 \fatsemi P_2$.

□

Condition (e) in Remark 4.1 leads to performing pattern matching using the concatenation operator. Therefore, we should add the assumption that the concatenation operator be left associative. In practice, however, this is unimportant since $(P_1 \fatsemi P_2) \fatsemi P_3 = P_1 \fatsemi (P_2 \fatsemi P_3)$.

4.2 Structural Induction Principle

We can now formulate a structural induction principle for annotated programs that does not refer to annotated statements.

4.2 Induction Principle for Annotated Programs Let $\mathcal{P}(P)$ be a property of annotated programs and assume that the following five conditions are satisfied.

a) $\mathcal{P}(\langle A \rangle)$ is true for all $A \in \mathbf{Annot}$.

b) $\mathcal{P}(\langle A_s \rangle \, \texttt{skip} \, \langle A_f \rangle)$ is true, for all $A_s, A_f \in \mathbf{Annot}$.

c) $\mathcal{P}(\langle A_s \rangle \, x := E \, \langle A_f \rangle)$ is true, for all variables $x \in \mathbf{Var}$, expressions $E \in \mathbf{Expr}$, and $A_s, A_f \in \mathbf{Annot}$.

d) $\mathcal{P}(\langle A_s \rangle \, \texttt{if} \; C \; \texttt{then} \; P_1 \; \texttt{else} \; P_2 \; \texttt{endif} \, \langle A_f \rangle)$ is true, for all $C \in \mathbf{Constr}$, $A_s, A_f \in \mathbf{Annot}$, and $P_1, P_2 \in \mathbf{AProg}(\mathbf{Annot})$ such that $\mathcal{P}(P_1)$ and $\mathcal{P}(P_2)$ both hold.

e) $\mathcal{P}(\langle A_s \rangle \, \texttt{while} \; C \; \texttt{do} \; P' \; \texttt{endwhile} \langle A_f \rangle)$ is true for all $C \in \mathbf{Constr}$, $A_s, A_f \in \mathbf{Annot}$, and $P' \in \mathbf{AProg}(\mathbf{Annot})$ such that $\mathcal{P}(P')$ holds.

f) $\mathcal{P}(P_1 \, ; P_2)$ is true, for all $P_1, P_2 \in \mathbf{AProg}(\mathbf{Annot})$ such that the last annotation of P_1 and the first annotation of P_2 are the same, and $\mathcal{P}(P_1)$ and $\mathcal{P}(P_2)$ both hold.

Then, $\mathcal{P}(P)$ holds for every annotated program P. $\square$

4.3 Annotations and Labels

Given a distinguished set **Labels** of labels, we represent imperative programs as annotated programs from $\mathbf{AProg}(\mathbf{Labels}) \backslash \{\langle l \rangle \mid l \in \mathbf{Labels}\}$ [2]. That is, we use labels as annotations in order to identify program points. The set of labels could be the set of natural numbers, as illustrated in Figure 4.2. We shall call annotated programs whose annotations are labels *labeled programs*, or simply *programs*. Given a labeled program P, we denote by *labels*(P) the set of labels occurring in P. Also,

[2] Single annotations are simply a notational convenience and do not represent real programs

we denote by *first*(P) the label associated with the first program point in P, and by *last*(P) the label associated with the last program point.

In order for the labeling to uniquely identify program points, the labels in a labeled program need to be distinct. We shall call a program with this property a *properly labeled program*. Formally, this property is expressed by the following conditions.

4.3 Proper Labeling Property If $l_s, l_f \in \mathbf{Labels}$ and $l_s \neq l_f$, then $\langle l_s \rangle \,\texttt{skip}\, \langle l_f \rangle$ and $\langle l_s \rangle\, x := E\, \langle l_f \rangle$ are properly labeled. Moreover, if $P_1, P_2 \in \mathbf{AProg}(\mathbf{Labels})$ such that $\mathit{labels}(P_1) \cap \mathit{labels}(P_2) = \emptyset$ and $l_s, l_f \notin \mathit{labels}(P_1) \cup \mathit{labels}(P_2)$, then $\langle l_s \rangle\, \texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{endif}\, \langle l_f \rangle$ and $\langle l_s \rangle\, \texttt{while}\ C\ \texttt{do}\ P_1 \texttt{endwhile} \langle l_f \rangle$ are properly labeled, for all $C \in \mathbf{Constr}$. Also, if $\mathit{labels}(P_1) \cap \mathit{labels}(P_2) = \{\mathit{last}(P_1)\} = \{\mathit{first}(P_2)\}$, then $P_1 \,;\, P_2$ is properly labeled. □

The proper labeling is a supplementary condition that needs to be added to the induction principle for labeled programs.

4.4 Induction Principle for Properly Labeled Programs Let $\mathcal{P}(P)$ be a property of labeled programs and assume that the following five conditions are satisfied simultaneously.

a) $\mathcal{P}(\langle l_s \rangle \,\texttt{skip}\, \langle l_f \rangle)$ is true, for all $l_s, l_f \in \mathbf{Labels}$ such that $l_s \neq l_f$.

b) $\mathcal{P}(\langle l_s \rangle\, x := E\, \langle l_f \rangle)$ is true, for all variables $x \in \mathbf{Var}$, expressions $E \in \mathbf{Expr}$, and $l_s, l_f \in \mathbf{Labels}$ such that $l_s \neq l_f$.

c) $\mathcal{P}(\langle l_s \rangle\, \texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{endif}\, \langle l_f \rangle)$ is true, for all $C \in \mathbf{Constr}$, $l_s, l_f \in \mathbf{Labels}$, and $P_1, P_2 \in \mathbf{AProg}(\mathbf{Labels})$ such that P_1, P_2 are properly labeled, $\mathit{labels}(P_1) \cap \mathit{labels}(P_2) = \emptyset$, $\{l_s, l_f\} \cap (\mathit{labels}(P_1) \cup \mathit{labels}(P_2)) = \emptyset$, and $\mathcal{P}(P_1)$ and $\mathcal{P}(P_2)$ both hold.

d) $\mathcal{P}(\langle l_s \rangle \texttt{ while } C \texttt{ do } P' \texttt{ endwhile} \langle l_f \rangle)$ is true for all $C \in \mathbf{Constr}$, $l_s, l_f \in \mathbf{Labels}$, and $P' \in \mathbf{AProg}(\mathbf{Labels})$ such that P' is properly labeled, $\{l_s, l_f\} \cap \mathit{labels}(P') = \emptyset$, and $\mathcal{P}(P')$ holds.

e) $\mathcal{P}(P_1 \,;\, P_2)$ is true, for all $P_1, P_2 \in \mathbf{AProg}(\mathbf{Labels})$ such that P_1 and P_2 are properly labeled, $\mathit{labels}(P_1) \cap \mathit{labels}(P_2) = \{\mathit{last}(P_1)\} = \{\mathit{first}(P_2)\}$, and $\mathcal{P}(P_1)$ and $\mathcal{P}(P_2)$ both hold.

Then, $\mathcal{P}(P)$ holds for every labeled program P. □

In what follows we shall be concerned only with properly labeled programs, and whenever we consider a labeled program, we shall assume that it is properly labeled.

Finally, we introduce some more terminology for labeled programs. Let P be a labeled program. Remark 4.1 allows the distinction between the following categories of annotated programs.

a) P is a *skip statement* if there exist $l_s, l_f \in \mathbf{Labels}$ such that $P = \langle l_s \rangle \texttt{ skip } \langle l_f \rangle$.

b) P is an *assignment* if there exist $l_s, l_f \in \mathbf{Labels}$, $x \in \mathbf{Var}$ and $E \in \mathbf{Expr}$ such that $P = \langle l_s \rangle\, x := E\, \langle l_f \rangle$.

c) P is a *sequence program* if there exist $P_1, P_2 \in \mathbf{AProg}(\mathbf{Labels})$ such that $\mathit{last}(P_1) = \mathit{first}(P_2)$ and $P = P_1 \,;\, P_2$. We call P_1 and P_2 the *first component* and the *second component* of the sequence program P, respectively.

d) P is an `if` *program* if there exist $l_s, l_f \in \mathbf{Labels}$, $C \in \mathbf{Constr}$ and $P_1, P_2 \in \mathbf{AProg}(\mathbf{Labels})$ such that $P = \langle l_s \rangle \texttt{ if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ endif} \langle l_f \rangle$. We call P_1 and P_2 the *consequent* and the *alternative* of the `if` program, respectively. Also, we call C the *if condition*.

e) P is a *while program* if there exist $l_s, l_f \in$ **Labels**, $C \in$ **Constr** and $P' \in$ **AProg**(**Labels**) such that $P = \langle l_s \rangle$ `while` C `do` P' `endwhile` $\langle l_f \rangle$. We call P' the *body* of the `while` program P. Also, we call C the *while condition*.

Finally, we introduce the notion of *program fragment* of a program P. If P is a sequence program, then both components of P are program fragments of P. Also, if P is an `if` program, then both the consequent and the alternative are program fragments of P, and if P is `while` program, the body of P is a program fragment of P. Finally, a program fragment of a program fragment of P is a program fragment of P.

Chapter 5

Operational, Trace, and Collecting Semantics

In this chapter we provide a brief overview of the operational, trace, and collecting semantics.

5.1 Operational Semantics

We present here an approach that is a slight variation of the small step (or structured) operational semantics proposed by Hennessy and Plotkin [HP79] and further developed by Plotkin [Plo81, AO97, Rey98]. The idea of Hennessy and Plotkin is to specify a transition relation $\xrightarrow{P}$, by induction on the structure of the program P using a formal proof system, called a *transition system*, and which consists of rules about transitions between states (called *configurations* in [Plo81]). Plotkin's configuration[1] is a pair $\langle P, \sigma \rangle$, consisting of a program P and an environment σ. Intuitively, a transition $\langle P_1, \sigma_1 \rangle \longrightarrow \langle P_2, \sigma_2 \rangle$ means: executing P_1 one step in an environment σ_1 can lead to an environment σ_2, with P_2 being the remainder of P_1

[1] In this thesis the word *configuration* has a different meaning

still to be executed.

We alter the transition system defined in [Plo81] by replacing the program component of a state with a label. The label has a role similar to the program counter of a processor, in the sense that it stores in a state the program point that has been reached after the last execution step. The advantage of using this approach shall become apparent when we introduce conditional reasoning in Chapter 15. There, the program point labels become part of the assertion language in order to allow tracking program point dependencies during propagation.

Before presenting the operational semantics of our language, we introduce some terminology. An *environment* is a mapping from variables to values. We distinguish between scalar values, which are integers, and array values, which are mappings from natural numbers to integers. We denote environments by the Greek letter σ, possibly subscripted, and the set of all environments by **Env**, i.e. $\mathbf{Env} = \{\sigma \mid \sigma : \mathbf{Var} \mapsto \mathbf{Values}\}$. We denote sets of environments by the symbol Σ.

A *state* is a pair $\langle l, \sigma \rangle \in \mathbf{Labels} \times \mathbf{Env}$. The set of all states $\mathbf{Labels} \times \mathbf{Env}$ is denoted by **States**, and the set $\mathit{labels}(P) \times \mathbf{Env}$ of all states of a program P is denoted $\mathit{States}(P)$. Moreover, we denote states by the letter s, and sets of states by the letter S, possibly subscripted. If $s = \langle l, \sigma \rangle$ is a state, we say that l is the *label of* s and σ is the *environment of* s.

A *transition system* is a pair (S, τ), where S is a non-empty set of states and $\tau \subseteq S \times S$ is a binary transition relation between a state and its possible successors. If $s \, \tau \, s'$ holds for some $s, s' \in S$, we say that there exists a transition from s to s'. If for every state $s \in S$ there exists at most one state s' such that $s \, \tau \, s'$, we say that the transition system (S, τ) is *deterministic*. We also denote by τ^* the transitive and reflexive closure of a relation τ.

In order to define a semantics for our programming language, we need to inter-

pret expressions and constraints of the language as values. We shall regard expressions $E \in \mathbf{Expr}$ as mappings from environments to values, and we shall write $E(\sigma)$ as the value of expression E in the environment σ. Similarly, we regard constraints $C \in \mathbf{Constr}$ as mappings from environments to truth values. We write $\sigma \models C$ to denote that the constraint C is true in environment σ.

Syntax-Directed Transition Relations

We shall now proceed with defining the operational semantics as a transition system $\mathcal{T}_P = (\mathit{States}(P), \xrightarrow{P})$ whose transitions describe the operational behavior of a program P. The $\xrightarrow{P}$ relation provides the transitions from the current state of executing the program to the next state and shall be defined inductively on the structure of the program P. For convenience, we shall express the relation $\xrightarrow{P}$ as the union of several sub-relations, that we define below. First, let P be an `if` program, and denote by C the `if` condition, and by P_c and P_a the consequent and the alternative of P, respectively. We define:

$$\begin{aligned}
\xrightarrow{P\swarrow} &\triangleq \{(\langle \mathit{first}(P), \sigma\rangle, \langle \mathit{first}(P_c), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models C\} \\
\xrightarrow{P\nearrow} &\triangleq \{(\langle \mathit{last}(P_c), \sigma\rangle, \langle \mathit{last}(P), \sigma\rangle) \mid \sigma \in \Sigma\} \\
\xrightarrow{P\searrow} &\triangleq \{(\langle \mathit{first}(P), \sigma\rangle, \langle \mathit{first}(P_a), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models \neg C\} \\
\xrightarrow{P\nwarrow} &\triangleq \{(\langle \mathit{last}(P_a), \sigma\rangle, \langle \mathit{last}(P), \sigma\rangle) \mid \sigma \in \Sigma\}
\end{aligned}$$

Next, let P be a `while` program, and denote by C the while condition, and by P_b the body of P. We define:

$$\begin{aligned}
\xrightarrow{P\searrow} &\triangleq \{(\langle \mathit{first}(P), \sigma\rangle, \langle \mathit{first}(P_b), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models C\} \\
\xrightarrow{P\nwarrow} &\triangleq \{(\langle \mathit{first}(P_b), \sigma\rangle, \langle \mathit{first}(P), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models \neg C\} \\
\xrightarrow{P\circlearrowleft} &\triangleq \{(\langle \mathit{last}(P_b), \sigma\rangle, \langle \mathit{first}(P_b), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models C\} \\
\xrightarrow{P\curvearrowright} &\triangleq \{(\langle \mathit{first}(P), \sigma\rangle, \langle \mathit{last}(P), \sigma\rangle) \mid \sigma \in \Sigma \text{ and } \sigma \models \neg C\}
\end{aligned}$$

Figure 5.1 depicts a representation of these relations. For an `if` program, as shown in Figure 5.1b, $\xrightarrow{P\swarrow}$ represents transitions between the start program point of the `if` construct and the start program point of its consequent, $\xrightarrow{P\nearrow}$ represents transitions between the program point at the end of the consequent and the last program point of the `if` construct, $\xrightarrow{P\searrow}$ represents transitions between the start program point of the `if` construct and the start program point of its alternative, while $\xrightarrow{P\nwarrow}$ represents transitions between the program point at the end of the consequent and the last program point of the `if` construct. The four sub-relations for a `while` program are depicted in Figure 5.1c. $\xrightarrow{P\searrow}$ represents transitions between the start of the `while` construct and the start of its body, $\xrightarrow{P\nwarrow}$ represents transitions between the end of the body and the end of the `while` construct, $\xrightarrow{P\circlearrowleft}$ represents transitions between the end and the beginning of the body (around-the-loop transition), while $\xrightarrow{P\curvearrowright}$ represent transitions between the beginning and the end of the `while` construct.

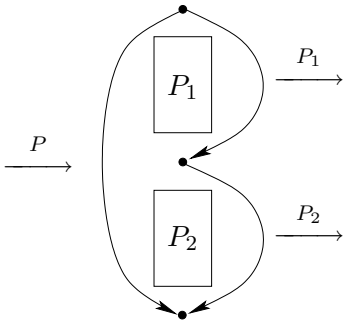
We define now the relation $\xrightarrow{P}$ recursively as follows:

a) If P is the skip statement $\langle l_s \rangle \,\texttt{skip}\, \langle l_f \rangle$, where $l_s, l_f \in \textbf{Labels}$, then $\xrightarrow{P}$ $= \{(\langle l_s, \sigma \rangle, \langle l_f, \sigma \rangle) \,|\, \sigma \in \textbf{Env}\}$.

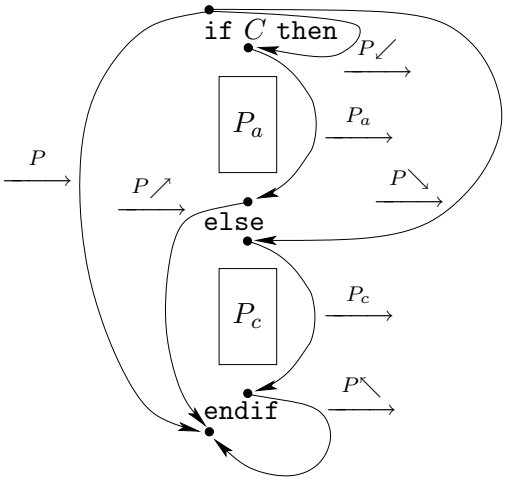
b) If P is the assignment $\langle l_s \rangle \, x := E \, \langle l_f \rangle$, where $l_s, l_f \in \textbf{Labels}$, $x \in \textbf{Var}$ and $E \in \textbf{Expr}$, then $\xrightarrow{P} = \{(\langle l_s, \sigma_s \rangle, \langle l_f, \sigma_f \rangle) \,|\, \sigma_s, \sigma_f \in \textbf{Env}$ and $\sigma_f = \sigma_s[x \mapsto E(\sigma_s)]\}$.

c) If P is an `if` statement, with P_1 and P_2 being the consequent and the alternative, then $\xrightarrow{P} = \xrightarrow{P\swarrow} \cup \xrightarrow{P\nearrow} \cup \xrightarrow{P\searrow} \cup \xrightarrow{P\nwarrow} \cup \xrightarrow{P_1} \cup \xrightarrow{P_2}$.

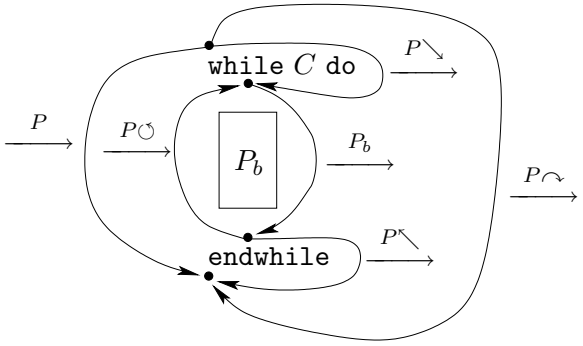
d) If P is a `while` statement, with P' being the body, then $\xrightarrow{P} = \xrightarrow{P\searrow}$ $\cup \xrightarrow{P\nwarrow} \cup \xrightarrow{P\circlearrowleft} \cup \xrightarrow{P\curvearrowright} \cup \xrightarrow{P'}$.



a) Sequence programs



b) `if` programs



c) `while` programs

Figure 5.1: Relationship between transition system and syntax

e) If P is the sequence statement $P_1 \,;\, P_2$, then $\xrightarrow{P} = \xrightarrow{P_1} \cup \xrightarrow{P_2}$.

Figure 5.1 shows a graphic representation of this definition for sequence, `if`, and `while` programs.

Transition System

5.1 Definition The *operational semantics* of a labeled program P is the transition system $(\mathit{labels}(P) \times \Sigma, \xrightarrow{P})$. □

The operational semantics defined above is a *high level* semantics, in the sense that the `skip` statement, assignments and evaluations of boolean expressions are all executed in one step and thus it abstracts from all details of the evaluation of expressions in the execution of assignments.

Given a state s of a labeled program P, if there is no state s' of P such that $s' \xrightarrow{P} s$, we say that s is an *initial state of* P. If there exists no state s' of P such that $s \xrightarrow{P} s'$, then we say that s is a *terminal state of* P.

Determinism

The following proposition shows that programs are deterministic, and have single entry and exit points, and will be useful in proving properties that relate traces to the syntax of the program.

5.2 Proposition Let P be a labeled program. The following statements hold.

a) $\langle \mathit{first}(P), \sigma \rangle$ is an initial state of P, for all environments $\sigma \in \Sigma$.

b) $\langle \mathit{last}(P), \sigma \rangle$ is a terminal state of P, for all environments $\sigma \in \Sigma$.

c) $(\mathbf{States}(P), \xrightarrow{P})$ is a deterministic transition system.

Proof: Relegated to Appendix A, on page 270.

5.2 Trace Semantics

In the previous section we have introduced a small step operational semantics for our programming language. The operational semantics is given as a transition system which specifies transitions from the current state to the next state of the program at hand. In what follows we shall define several more semantics, with various levels of abstraction, and argue that each of these semantics favors a specific kind of program reasoning. Since the reasoning framework that we develop is based on the notion of *propagation*, it is useful to specify these semantics in a fixpoint manner, and then define the propagation operator as an abstraction of the semantic transformer of the semantics.

In this section we shall define the *trace semantics*, which is the set of all execution sequences (i.e. sequences of states) that occur in all the possible executions of a program with respect to a given set of *start states*. Traces have been used to specify the semantics of both programming languages [Hoa78, AO97] and modal logics [Kri63]. Cousot [Cou02] distinguishes between three kinds of trace semantics.

- The *partial execution trace semantics* is the set of all execution sequences that may occur during the execution of the program.
- The *maximal finite trace semantics* is the set of all finite execution sequences that are complete (i.e. that cannot be extended).
- The *maximal trace semantics* is the set of all (i.e. finite and infinite) complete execution sequences.

Each of these definitions have specific advantages. The maximal trace semantics

is able to capture non-termination and indicate the conditions under which infinite computations may occur. Maximal finite trace semantics is amenable to compositional reasoning. Partial execution trace semantics on the other hand, while being polluted by execution sequences that only reflect partial computations, has the advantage of a fixpoint specification that has a rather simple semantic transformer. This advantage, together with the fact that our program-point-based abstraction method would allow us to eliminate the polluting execution sequences later, makes the partial execution trace semantics the preferred one. For simplicity, in what follows we shall refer to a partial execution trace as simply a trace. We proceed now with the definition of the trace semantics.

Traces

Let P be a labeled program. A *trace of P starting in σ_0* is a finite sequence of states $s_0 s_1 \cdots s_{k-1} s_k$ such that $k > 0$, $s_0 = \langle \mathit{first}(P), \sigma_0 \rangle$, and whenever $k > 1$ we have $s_i \xrightarrow{P} s_{i+1}$, for all $i,\ 0 \leq i < k$.

5.3 Definition Let P be a labeled program. Given a set of start environments Σ_0, the *trace semantics* of P w.r.t. Σ_0 is the set of all traces $\langle l_0, \sigma_0 \rangle s_1 \cdots s_k \in \vec{P}$ such that $\sigma_0 \in \Sigma_0$. $\Box$

We denote by $\overset{\Sigma_0\rightarrow}{P}$ the trace semantics of a labeled program P w.r.t. a set of start environments Σ_0. Whenever the set of start environments is **Env**, we write $\vec{P}$ instead of $\overset{\mathbf{Env}\rightarrow}{P}$. We note that $\overset{\mathbf{Env}\rightarrow}{P} \subseteq \vec{P}$, for any set of start environments Σ_0.

We denote traces by the symbol θ, and sets of traces by the symbol Θ, possibly subscripted. Given a program P, a sequence of states $s_1 s_2 \cdots s_k$ with the property that $s_i \xrightarrow{P} s_{i+1}$, $1 \leq i < k$, whenever $k \geq 2$, is called a *trace segment*. The empty sequence ϵ, and a singleton sequence s are also trace segments. We denote

trace segments by the letter t. The concatenation of two trace segments t_1 and t_2 is denoted $t_1 t_2$. If $t_1 = s_1 \cdots s_m$ and $t_2 = s_1 \cdots s_n$, with $m \leq n$, then we say that t_1 is a prefix of t_2. If $m < n$, then t_1 is a *proper prefix* of t_2. In this case, s_{m+1} is the *state following* t_1 *in* t_2. If $m = n - 1$, then t_1 is the *longest proper prefix* of t_2. The empty sequence ϵ is a proper prefix of any non-empty trace segment. Given two trace segments t_1 and t_2, we denote by $t_1 \xrightarrow{P} t_2$ the fact that there exists a transition from the last state of t_1 to the first state of t_2. Finally, we note that the notions of trace and trace segment can be defined for any transition system, and not only for programs. We shall take advantage of this observation in the next section.

Trace Pattern Matching

Trace pattern matching is a useful tool in representing the inductive structure of a trace segment with respect to a program fragment.

Given a program P, a *trace pattern matching expression* is either

a) a state of P, or

b) an expression of the form $\underbrace{t}_{P'}$, where t is a trace segment variable and P' is a program fragment of P, or

c) an expression of the form $PM_1 \, \tau \, PM_2$, where PM_1 and PM_2 are trace pattern matching expressions, and τ is a sub-relation of $\xrightarrow{P}$.

Given a program P, a trace segment t, and a pattern matching expression PM, we say that t *matches* PM, and we write $t \sim PM$ if either of the following conditions holds:

a) PM is a state s, and t is the same state s, or

b) PM is the expression $\underbrace{t'}_{P'}$, and t is a trace of program P' where P' is a program fragment of P, and t' is a trace segment variable that matches t, or

c) PM is the expression $PM_1 \tau PM_2$, and t can be written as $t_1 t_2$, where PM_1 and PM_2 are trace pattern matching expressions, t_1 and t_2 are trace segments such that $t_1 \sim PM_1$ and $t_2 \sim PM_2$, τ is a sub-relation of $\xrightarrow{P}$, and there exists a transition $s_1 \tau s_2$ between the last state s_1 of t_1 and the first state s_2 of t_2.

For instance, consider an `if` program P whose condition is C and consequent is P_c, and assume that $\theta \in \overrightarrow{P}$ is a trace. Writing $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_1} \xrightarrow{P\nearrow} \langle \mathit{last}(P), \sigma_f \rangle$ means that there exist environments $\sigma_s, \sigma_f \in \mathbf{Env}$ and the trace segment $t \in \overrightarrow{P_1}$ such that the trace θ starts with the state $\langle \mathit{first}(P), \sigma_s \rangle$, and via a transition of $\xrightarrow{P\swarrow}$, continues with the trace segment t, which is in fact a trace of P_1. The trace segment t ends with a state whose label is $\mathit{last}(P_c)$, and then follows a the state $\langle \mathit{last}(P), \sigma_f \rangle$, via a transition of $\xrightarrow{P\nearrow}$. We note that the trace segment variable t is uniquely matched with the sub-segment of θ whose transitions belong to $\xrightarrow{P_c}$. The definition of pattern-matching expressions does not ensure that trace segment variables are uniquely matched. However, we shall always employ only expressions that lead to unique matching of variables with trace segments.

The following proposition is the consequence of our programming language being deterministic and shall be useful in providing a syntax-based representation of traces.

5.4 Proposition Let P be a labeled program and $\theta_1, \theta_2 \in \overrightarrow{P}$ two traces with the same start states. Then, either θ_1 is a prefix of θ_2, or θ_2 is a prefix of θ_1.

Proof: The proof is by induction on the length of both θ_1 and θ_2. The base case is when both traces have a length that is equal to 1. Then, $\theta_1 = \theta_2 = \langle l, \sigma \rangle$, where $l = \mathit{first}(P)$ and $\sigma \in \mathbf{Env}$. In this case, the proposition is trivially true.

For the induction case, assume that θ_1 and θ_2 have lengths that are less than or equal to n. We have that $\theta_1 = \theta_1' \langle l_1, \sigma_1 \rangle$ and $\theta_2 = \theta_2' \langle l_2, \sigma_2 \rangle$, where $\theta_1', \theta_2' \in \vec{P}$ are traces of length less than n, $l_1, l_2 \in \mathit{labels}(P)$ and $\sigma_1, \sigma_2 \in \mathbf{Env}$. From the induction hypothesis, θ_1' is a prefix of θ_2', or θ_2' is a prefix of θ_1'. Assume that θ_1' is a prefix of θ_2' (the case when θ_2' is a prefix of θ_1' is proved in a similar way). Since θ_1' is a prefix of θ_2', θ_1' is a prefix of θ_2. From $\theta_1 = \theta_1' \langle l_1, \sigma_1 \rangle \in \vec{P}$ we have that $\theta_1' \xrightarrow{P} \langle l_1, \sigma_1 \rangle$. According to Proposition 5.2, given θ_1, $\langle l_1, \sigma_1 \rangle$ is unique. Therefore, $\langle l_1, \sigma_1 \rangle$ is the state following θ_1' in θ_2. As a result, $\theta_1' \langle l_1, \sigma_1 \rangle$ is a prefix of θ_2, which proves the inductive case. □

We shall now provide a fixed point characterization of the trace semantics.

5.3 The Trace Progress Operator

Following [Cou02] we will provide a fixpoint characterization of the trace semantics. First we remark that $2^{(\mathbf{Labels} \times \mathbf{Env})^*}$, which is the domain of the trace semantics, is a complete lattice. We now introduce a semantic transformer that takes each trace of a set and extends it by exactly one state.

5.5 Definition Given a labeled program P, we define the *trace progress operator* $\vec{T}_P$ as follows:

$$\vec{T}_P(\Theta) = \{ s_1 s_2 \cdots s_k s_{k+1} \mid s_1 s_2 \cdots s_k \in \Theta \text{ and } s_k \xrightarrow{P} s_{k+1} \}$$

□

We intend to prove that the least fixed point of $\vec{T}_P$ is the trace semantics for all labeled programs P. The following proposition shows that $\vec{T}_P$ has a least fixed point.

5.6 Proposition For any labeled program P, $\vec{T}_P$ is continuous.

Proof: We need to prove that for any directed subset $X \subseteq 2^{(\mathbf{Labels}\times\mathbf{Env})^*}$, $\vec{T}_P(lub(X)) = lub(\{\vec{T}_P(A) \mid A \in X\})$. We will prove in fact a stronger condition, that the above equality holds for all $X \subseteq 2^{(\mathbf{Labels}\times\mathbf{Env})^*}$, and not only for the directed ones.

Assume we have a trace $s_1 s_2 \cdots s_i s_{i+1} \in \vec{T}_P(lub(X))$, for some $i \geq 0$. This is equivalent to $s_0 s_1 \cdots s_i \in lub(X)$ and $s_i \xrightarrow{P} s_{i+1}$, which is in turn equivalent to the fact that there exists some $A \in X$ such that $s_0 s_1 \cdots s_i \in A$. Therefore, $s_0 s_1 \cdots s_i s_{i+1} \in \vec{T}_P(A)$. This entails that $s_0 s_1 \cdots s_i s_{i+1} \in lub(\{\vec{T}_P(A) \mid A \in X\})$. $\square$

Fixpoint Characterization

We can now provide a fixed point characterization of the trace semantics.

5.7 Proposition Let P be a program, and Σ_0 a set of start environments. Let $\Theta_0 = \{\langle first(P), \sigma_0\rangle \mid \sigma_0 \in \Sigma_0\}$. Then, $lfp(\vec{T}_P \cup \Theta_0) = \overset{\Sigma_0\rightarrow}{P}$.

Proof: We first prove that $\overset{\Sigma_0\rightarrow}{P} \subseteq lfp(\vec{T}_P \cup \Theta_0)$.

Since $lfp(\vec{T}_P \cup S_0) = (\vec{T}_P \cup S_0) \uparrow \omega = \bigcup_{n\geq 0}(\vec{T}_P \cup S_0) \uparrow n$, we need to show that for every trace t in the trace semantics, there exists $n \in \mathbb{N}$ such that $t \in (\vec{T}_P \cup S_0) \uparrow n$.

We prove by induction that every trace of length n is a member of $(\vec{T}_P \cup S_0) \uparrow n$. The base case is trivially true. For the induction case, assume we have a trace $t = s_1 s_2 \cdots s_n s_{n+1}$ and we want to prove that $t \in (\vec{T}_P \cup S_0) \uparrow (n+1)$. Since t

is a trace, $s_1 s_2 \cdots s_n$ is also a trace, and according to the induction hypothesis, $s_1 s_2 \cdots s_n \in (\vec{T}_P \cup S_0) \uparrow n$. The fact that $s_1 s_2 \cdots s_n s_{n+1}$ is a trace implies $s_n \xrightarrow{P} s_{n+1}$, which entails that $s_1 s_2 \cdots s_n s_{n+1} \in (\vec{T}_P \cup S_0)(\{s_1 s_2 \cdots s_n\})$. Due to the monotonicity of $\vec{T}_P \cup S_0$, it follows that $s_1 s_2 \cdots s_n s_{n+1} \in (\vec{T}_P \cup S_0)((\vec{T}_P \cup S_0) \uparrow n)$.

We now show that $lfp(\vec{T}_P \cup \Theta_0) \subseteq \overset{\Sigma_0\rightarrow}{P}$. We prove by induction that $(\vec{T}_P \cup \Theta_0) \uparrow n \subseteq \overset{\Sigma_0\rightarrow}{P}$ for all $n \in \mathbb{N}$. For $n = 0$ the statement is trivially true. For the induction case, consider a trace $\theta \in (\vec{T}_P \cup \Theta_0) \uparrow n$. If $\theta = \epsilon$, then the statement is trivially true. Otherwise, denote by θ' and by s the longest proper prefix and the last state of θ, respectively. By the definition of $\vec{T}_P$ given in Definition 5.5, θ' must be in $(\vec{T}_P \cup \Theta_0) \uparrow (n-1)$ and $\theta' \xrightarrow{P} s$ must hold. According to the induction hypothesis, $\theta' \in \overset{\Sigma_0\rightarrow}{P}$. This entails that $\theta = \theta' s \in \overset{\Sigma_0\rightarrow}{P}$. Since θ has been chosen arbitrarily as a member of $(\vec{T}_P \cup \Theta_0) \uparrow n$, then $(\vec{T}_P \cup \Theta_0) \uparrow n \subseteq \overset{\Sigma_0\rightarrow}{P}$, which proves the induction case. □

Next, we focus our attention to the collecting semantics, which is the basis for most classic program analysis frameworks.

5.4 Collecting Semantics

The trace semantics defined in Section 5.2 provides the behavior of a program in terms of the set of sequences of states that occur during the execution of the program. However, due to its high level of detail, this view of the trace semantics dilutes meaningful properties that are true of the execution of the program. For example, it does not describe the values a program variable may take at a specific program point during execution, or whether the execution terminates or not. Such information is clearly central to program reasoning. What is required, therefore, is a semantics that *exposes* interesting properties about the execution of the program.

One way to achieve this is to collect in a set the environments encountered at each program point (label) during program execution. This set is called the *collecting semantics*.

The notion of collecting semantics is the starting point of many formal treatments of program reasoning techniques, including program analysis and program verification. The concept has been introduced by Cousot [CC79b]. Our definition of the collecting semantics is just an explication of information already implicit in the trace semantics. In essence, the trace semantics is projected onto the notion of program point. As an aside, we note that since a collecting semantics describes what happens part way through a computation (including computations that lead to an error or do not terminate), a natural semantics style presentation of the operational semantics [Kah87, Mel85] would be significantly less convenient than the transition system style we have employed.

Abstraction by Sets of States

Before presenting the definition of the collecting semantics, we address the issue of starting environments. In the operational semantics, it was appropriate to define program execution from some given starting environment σ_0. However, when reasoning about a program, the initial environment may not be known. This issue may be addressed in a number of ways. First, program execution could be defined to start in a fixed initial environment (which, say, maps all variables to 0). Second, programs could be defined to begin with a sequence of assignment statements that initialize all program variables, and then the initial environment is essentially irrelevant. Third, collecting semantics could be defined to be the environments encountered over executions from all possible starting environments. Fourth, the collecting semantics could be defined with respect to a set Σ_0 of starting environ-

ments. Of these four possibilities, the last two are the most reasonable. We choose the last one, since it also enjoys the property of being compositional, i.e. of allowing the collecting semantics of several program fragments to be combined in a syntax-based manner in order to produce the collecting semantics of a larger program. The definition of the collecting semantics of an imperative program P can now be presented.

5.8 Definition Let P be a labeled program and Σ_0 a set of start environments, and denote by S_0 the set $\{\langle \mathit{first}(P), \sigma_0 \rangle \,|\, \sigma_0 \in \Sigma_0\}$ The *collecting semantics* of P w.r.t. Σ_0 is the set

$$\{s \,|\, \text{there exists } s' \in S_0 \text{ such that } s' \xrightarrow{P}{}^{*} s\}$$

□

Figure 5.2 shows a labeled program and its collecting semantics.

A definition of collecting semantics is the starting point of classical program analysis. In particular it provides the primary definition of correctness: an (approximate) analysis is correct if it yields a *conservative approximation* of the collecting semantics. In other words, an analysis is correct if, for each program point, the set of environments described by the analysis for that point is a superset of the set of environments described by the collecting semantics. As we will show later, this definition of correctness conceals properties like termination and liveness. To overcome this, we shall introduce the progressive semantics of an imperative program and a new definition of correctness that would be amenable to proving liveness and termination properties, in addition to the conservative properties that are supported by the collecting semantics.

$\langle 1 \rangle$
  $x := 0$
$\langle 2 \rangle$
  $y := 0$
$\langle 3 \rangle$
  `while` $x < 10$ `do`
    $\langle 4 \rangle$
      $x := x + 1$
    $\langle 5 \rangle$
      $y := y + x$
    $\langle 6 \rangle$
  `endwhile`
$\langle 7 \rangle$

(a) A labeled program

$$
\begin{aligned}
\{\ & \langle 1, [x \mapsto 0, y \mapsto 0] \rangle, \langle 1, [x \mapsto 1, y \mapsto 0] \rangle, \langle 1, [x \mapsto 0, y \mapsto 1] \rangle, \ldots, \\
& \langle 2, [x \mapsto 0, y \mapsto 0] \rangle, \langle 2, [x \mapsto 0, y \mapsto -1] \rangle, \langle 2, [x \mapsto 0, y \mapsto 1] \rangle, \ldots, \\
& \langle 3, [x \mapsto 0, y \mapsto 0] \rangle, \\
& \langle 4, [x \mapsto 0, y \mapsto 0] \rangle, \langle 4, [x \mapsto 1, y \mapsto 1] \rangle, \cdots, \langle 4, [x \mapsto 9, y \mapsto 45] \rangle, \\
& \langle 5, [x \mapsto 1, y \mapsto 0] \rangle, \langle 5, [x \mapsto 2, y \mapsto 1] \rangle, \cdots, \langle 5, [x \mapsto 10, y \mapsto 45] \rangle, \\
& \langle 6, [x \mapsto 1, y \mapsto 1] \rangle, \langle 6, [x \mapsto 2, y \mapsto 3] \rangle, \cdots, \langle 6, [x \mapsto 10, y \mapsto 55] \rangle, \\
& \langle 7, [x \mapsto 10, y \mapsto 55] \rangle \ \}
\end{aligned}
$$

(b) The collecting semantics represented as a set of states

Figure 5.2: The Collecting Semantics of a Program

Transfer Function

We introduce a semantic transformer which, given a set of "current" states, computes a set of "next" states.

5.9 Definition Given a program P, the *transfer function* $T_P : 2^{labels(P)\times\mathbf{Env}} \mapsto 2^{labels(P)\times\mathbf{Env}}$ is defined as

$$T_P(\Sigma) = \{\langle n, \sigma\rangle \,|\, \text{there exists } \langle n', \sigma'\rangle \in \Sigma \text{ such that } \langle n', \sigma'\rangle \xrightarrow{P} \langle n, \sigma\rangle\}$$

□

The structure $(2^{labels(P)\times\mathbf{Env}}, \subseteq)$ is clearly a lattice. In order to show that the transfer function provides a fixpoint semantic specification, we need to prove the following two propositions.

5.10 Proposition For all labeled programs P the operator T_P is continuous.

Proof: We need to prove that for every directed set $X \subseteq 2^{\mathbf{Labels}\times\mathbf{Env}}$, $T_P(lub(X)) = lub(\{T_P(A) \,|\, A \in X\})$. We will prove in fact a stronger condition, that the above equality holds for any $X \subseteq 2^{\mathbf{Labels}\times\mathbf{Env}}$. We proceed by proving that for all s, $s \in T_P(lub(X))$ is equivalent to $s \in lub(\{T_P(A) \,|\, A \in X\})$.

Let $s \in T_P(lub(X))$. This is equivalent with the fact that there exists a state $s' \in lub(X)$ such that $s \in T_P(\{s'\})$. The fact that $s' \in lub(X)$ is equivalent with the fact that there exists a set of states $A \in X$ such that $s' \in A$. It follows immediately that $s \in T_P(\{s'\}) \subseteq T_P(A)$, which is equivalent to $s \in lub(\{T_P(A) \,|\, A \in X\})$. □

5.5 Fixpoint Characterization

5.11 Proposition The collecting semantics of a program P and a set of start

states S_0 is the least fixpoint of $T_P \cup S_0$.

Proof: Denote by *CS* the collecting semantics of program P. We first prove that $CS \subseteq lfp(T_P \cup S_0)$.

Since $lfp(T_P \cup S_0) = (T_P \cup S_0) \uparrow \omega = \bigcup_{n \geq 0}(T_P \cup S_0) \uparrow n$, we need to show that for every state s in the collecting semantics, there exists $n \in \mathbb{N}$ such that $s \in (T_P \cup S_0) \uparrow n$.

We prove by induction that, given a start state $s_1 \in S_0$, every state s at the end of a trace $s_1 s_2 \cdots s_{n-1} s$ of length n is a member of $(T_P \cup S_0) \uparrow n$. The base case is trivially true. For the induction case, assume we have a state s which is at the end of a trace $s_1 s_2 \cdots s_{n-1} s$. According to the induction hypothesis, $s_{n-1} \in (T_P \cup S_0) \uparrow (n-1)$. Also, from the fact that $s_1 s_2 \cdots s_{n-1} s$ is a trace we have that $s \in (T_P \cup S_0)(\{s_{n-1}\})$. Since $T_P \cup S_0$ is monotonic, we have that $s \in (T_P \cup S_0)((T_P \cup S_0) \uparrow (n-1)) = (T_P \cup S_0) \uparrow n$.

Now, we prove that $lfp(T_P \cup S_0) \subseteq CS$. It is sufficient to prove that $(T_P \cup S_0) \uparrow n \subseteq CS$ for all $n \in \mathbb{N}$. The base case is again trivially true. For the induction case, consider a state $s \in (T_P \cup S_0) \uparrow n$. There must exist a state $s' \in (T_P \cup S_0) \uparrow (n-1)$ such that $s' \xrightarrow{P} s$. From the induction hypothesis, we have that there must exist a state $s_0 \in S_0$ such that $s_0 \xrightarrow{P}{}^* s'$, which in turn entails that $s_0 \xrightarrow{P}{}^* s$. Therefore, $s \in CS$, and since s is arbitrarily chosen, it results that $(T_P \cup S_0) \uparrow n \subseteq CS$. $\square$

In what follows, we shall provide syntactic structure to the trace and collecting semantics and introduce the progressive semantics in a syntax-directed manner as well. The first step in adding syntactic structure to the trace semantics is to abstract the sequencing information contained in a trace by projecting it on the labels of a

program, and this shall be achieved by means of indexed sets, which are merely mappings from indices to sets of environments.

Part II

Progressive Semantics

Chapter 6

Syntax-Based Semantic Representation

6.1 Indices and Indexed Sets

In this section we shall introduce the notion of *indexed set of environments* (or *indexed sets* for short), which provides a way to structure the information given by the collecting semantics such that it exposes a wider range of program properties, for instance liveness and progress. Essentially, an indexed set is a mapping from a set of ordered indices to the powerset of environments. The ordering of the indices will help us capture an abstraction of the sequence of environments that occur at a program point during the execution of the program, which in turn will allow us to prove inevitability properties inductively. Indexed sets shall be used as annotations in annotated programs, and the resulting structure, called a *progressive configuration* shall be used to express the progressive semantics of a labeled program.

An *index* is a sequence of natural numbers. We denote indices by $\tilde{\mu} = \mu_1\mu_2\cdots\mu_p$, where $\mu_i \in \mathbb{N}$, $1 \leq i \leq p$. We denote the set of indices by **Idx**. Indices will be

denoted by $\tilde{\mu}$, possibly subscripted, while components of an index will be denoted by μ_i, where $i \in \mathbb{N}$. If $\tilde{\mu}_1 = \mu_1\mu_2\cdots\mu_k$ and $\tilde{\mu}_2 = \mu_1\mu_2\cdots\mu_k\mu_{k+1}$ are two indices, we say that $\tilde{\mu}_1$ is the *longest proper prefix* of $\tilde{\mu}_2$, and we denote that by $\tilde{\mu}_1 = lpp(\tilde{\mu}_2)$. We also write $\tilde{\mu}_2 = \tilde{\mu}_1\mu_{k+1}$. We define the successor of $\tilde{\mu}_2$ as $\tilde{\mu}_1(\mu_{k+1}+1)$, and we denote it by $succ(\tilde{\mu}_2)$. The size k of $\tilde{\mu}_1$ is denoted by $size(\tilde{\mu}_1)$. Indices in **Idx** are ordered lexicographically.

An indexed set is a mapping from indices in **Idx** to sets of environments. Indexed sets shall be denoted by the Greek letter Ψ, and the set of all indexed sets shall be denoted by **IdxEnv**. Given an indexed set Ψ and an index $\tilde{\mu}$, we call $\Psi(\tilde{\mu})$ a *slice* of Ψ. An indexed set that maps every index into either a singleton or the empty set is called an *indexed singleton*. The set of all indexed singletons is denoted by **IdxSingleton**. A cpo over the domain of indexed sets can be easily defined by $\Psi_1 \subseteq \Psi_2$ if $\Psi_1(\tilde{\mu}) \subseteq \Psi_2(\tilde{\mu})$ for all $\tilde{\mu} \in$ **Idx**.

A special class of indexed sets deserves special attention. It is the class of all indexed sets Ψ for which there exists a natural number k such that $\Psi(\tilde{\mu}) = \emptyset$ for all $\tilde{\mu}$ such that $size(\tilde{\mu}) \neq k$. Such indexed sets shall be represented by formulas of the form $\lambda\langle\mu_1\cdots\mu_k\rangle\,.\,\varphi(\mu_1,\ldots,\mu_k)$, where $\mu_1,\cdots\mu_k$ are index variables, and φ is a set expression that depends on $\mu_1,\ldots,\mu_k$. The indexed set Ψ represented by this formula is defined by

$$\Psi(\tilde{\mu}) = \begin{cases} \emptyset\ , & \text{if } size(\tilde{\mu}) \neq k \\ \varphi(\mu_1,\ldots,\mu_k)\ , & \text{if } size(\tilde{\mu}) = k \text{ and } \tilde{\mu} = \mu_1\cdots\mu_k \end{cases}$$

For example, the formula $\Psi = \lambda\langle\mu_1\mu_2\rangle\,.\,\{\sigma \mid \sigma(x) = \mu_1 \text{ and } \sigma(y) = \mu_2\}$ denotes an indexed set such that, on one hand $\Psi(\tilde{\mu}) = \emptyset$ for all $\tilde{\mu}$ whose size is not equal to 2, and on the other hand, $\Psi(\mu_1\mu_2)$ is the set of environments that map variable x into μ_1 and variable y into μ_2. The indexed set that maps all indices into the empty set shall be denoted by $\lambda\langle\rangle\,.\,\emptyset$.

6.2 Configurations

Configurations are a syntax-directed means of expressing information about programs. We will show that both the trace semantics and the collecting semantics of a program can be expressed as configurations. Moreover, the progressive semantics that we introduce later in this chapter is also represented as a configuration. The main advantages of configurations are, on one hand, that they project traces or sets of states on labels in a natural way, thus making explicit what properties hold at a program point, and on the second hand, that they are compositional, due to their syntax directed definition. We implement configurations as annotated programs, by specializing the annotation set to be the cross-product between a set of labels and either the set of environments, or the set of indexed sets, or a set of formulas, as it shall be the case in the next chapter. In this chapter, we shall distinguish between three kinds of configurations:

a) *singleton configurations*, which are members of $\mathbf{AProg}(\mathbf{Labels} \times \mathbf{IdxSingleton})$;

b) *progressive configurations*, which are members of $\mathbf{AProg}(\mathbf{Labels} \times \mathbf{IdxEnv})$;

c) *collective configurations*, which are members of $\mathbf{AProg}(\mathbf{Labels} \times 2^{\Sigma})$.

We denote configurations with the letter K, possibly subscripted. To distinguish between singleton, progressive and collecting configurations, we denote singleton configurations by $\vec{K}$, progressive configurations by K, and collective configurations by $\overline{K}$. Whenever the type of configuration is not important, we shall simply use the letter K to denote it. Given a configuration K, we denote by $|K|$ the underlying labeled program of K. We also extend the notation *labels* to configurations, in the form $\mathit{labels}(K) = \mathit{labels}(|K|)$. We also extend the notations *first* and *last* in the

same manner, that is, $\mathit{first}(K) = \mathit{first}(|K|)$ and $\mathit{last}(K) = \mathit{last}(|K|)$. Moreover, given a label $l \in \mathit{labels}(K)$, we denote by $K|_l$ the program point property $\mathcal{P}$ in the annotation $\langle l, \mathcal{P}\rangle$ that occurs in configuration K at program point l. Finally, *sub-configuration* K' of a configuration K is a configuration that occurs inside K. Formally, K' is a sub-configuration of K if K is a sequence configuration and K' is one of its components, or if K is an *if* configuration and K' is either its consequent or its alternative, or if K is a *while* configuration and K' is its body, or if there exists a configuration K'' such that K'' is a sub-configuration of K and K' is a sub-configuration of K''.

6.1 Remark Let K be a configuration and K' a sub-configuration of K. If $l \in \mathit{labels}(K')$, then $K|_l = K'|_l$. □

Assuming that there exists a partial order $\leq$ on the domain of program point properties, we can easily define a partial order $\preccurlyeq$ over the a set of configurations with the same skeleton, by simply stating that for any two configurations K_1 and K_2, $K_1 \preccurlyeq K_2$ holds whenever $|K_1| = |K_2|$, and $K_1|_l \leq K_2|_l$, for all $l \in \mathit{labels}(K_1)$.

The next proposition shows that a set of configurations can be made into a lattice whenever the annotations are members of a lattice. This property shall be useful in arguing that configurations can be used in fixpoint semantic specifications.

6.2 Proposition Given a labeled program P and a complete lattice $(L, \leq)$, denote by Γ the set $\{K \mid K \in \mathbf{AProg}(\mathbf{Labels} \times L)$ and $|K| = P\}$, and by $\preccurlyeq$ the cpo induced by the $\leq$ relation on $\mathbf{AProg}(\mathbf{Labels} \times L)$. Then $(\Gamma, \preccurlyeq)$ is a complete lattice.

Proof: We have to show that for every set $X \subseteq \Gamma$ of configurations, $\mathit{lub}(X)$ and $\mathit{glb}(X)$ exist in Γ. Let X be an arbitrary subset of Γ and $l \in \mathit{labels}(P)$ an arbitrary label of P, and denote by X_l the set $\{(K|_l) \mid K \in X\}$. Clearly, $X_l \subseteq L$,

and since $(L, \leq)$ is a lattice, $lub(X_l)$ and $glb(X_l)$ exist in L. Denote now by K_{lub} a configuration such that $|K_{lub}| = P$ and for every label $l \in labels(P)$ we have $K_{lub}|_l = lub(\{(K|_l) \,|\, K \in X\})$. Also, denote by K_{glb} a configuration such that $|K_{glb}| = P$ and for every label $l \in labels(P)$ we have $K_{glb}|_l = glb(\{(K|_l) \,|\, K \in X\})$. It is immediate to prove that $K_{lub} = lub(X)$ and $K_{glb} = glb(X)$. Indeed, K_{lub} is an upper bound of X since $K_{lub}|_l \leq K|_l K$, for all $K \in X$ and $l \in labels(P)$. Moreover, given an upper bound K' of X, we have that $K \preccurlyeq K'$ for all $K \in X$, which entails $K|_l \leq K'|_l$, for all $K \in X$ and $l \in labels(P)$. Now, $K_{lub}|_l$ is the least upper bound for $\{(K|_l) \,|\, K \in X\}$, and therefore $K_{lub}|_l \leq K'|_l$ for all $l \in labels(P)$. It results that $K' \preccurlyeq K_{lub}$, which proves that $K_{lub} = lub(X)$. A similar argument can be made to prove that $K_{glb} = glb(X)$. $\square$

Chapter 7

Traces as Configurations

In what follows, we shall define a way to represent traces, the trace semantics and the collecting semantics as configurations. The first step is to establish a link between the syntactic structure of a program and its traces.

7.1 Compositional Traces

In this section we show that traces of a program are made up from the traces of its fragments. The following three propositions will help accomplishing this goal.

7.1 Proposition Let $P = P_1 \,;\, P_2$ be a sequence program and $\theta \in \overrightarrow{P}$ be a trace. Exactly one of the following statements holds:

a) $\theta \sim \underbrace{t}_{P_1}$

b) $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{\langle last(P_1), \sigma \rangle t_2}_{P_2}$

Proof: Relegated to Appendix A, on page 272.

7.2 Proposition Let P be an `if` program whose condition is C and consequent and alternative are P_c and P_a. Consider a trace $\theta \in \overrightarrow{P}$. Exactly one of the following statements is true:

a) $\theta = \langle \mathit{first}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$.

b) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models C$, and trace segment $t \in \overrightarrow{P_c}$.

c) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} \langle \mathit{last}(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models C$, and $t \in \overrightarrow{P_c}$.

d) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models \neg C$, and trace segment $t \in \overrightarrow{P_a}$.

e) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models \neg C$, and $t \in \overrightarrow{P_c}$.

Proof: Relegated to Appendix A, on page 273.

7.3 Proposition Let P be a `while` program whose condition is C and body is P_b. Consider a trace $\theta \in \overrightarrow{P}$. Exactly one of the following statements is true:

a) $\theta = \langle \mathit{first}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$.

b) $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\curvearrowright} \langle \mathit{last}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$, such that $\sigma \models \neg C$.

c) $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b}$, for some $k > 0$ and some environment $\sigma \in \mathbf{Env}$ such that $\sigma \models C$.

d) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, for some $k > 0$ and some environments $\sigma_s, \sigma_f \in \mathbf{Env}$ such that $\sigma_s \models C$ and $\sigma_f \models \neg C$.

Proof: Relegated to Appendix A, on page 274.

7.2 Progressive Segmentation

In this subsection we show that traces can be represented as singleton configurations. This shall provide a tighter link between trace semantics and the syntax of the program.

We start by defining nesting relations between labels. Let P be a labeled program and consider two labels $l_s, l_f \in \mathit{labels}(P)$. We say that l_s *leads to and is on the same level* as l_f, denoted $l_s \circledcirc l_f$, if there exists a program fragment P' of P such that either $l_s = \mathit{first}(P')$ and $l_f = \mathit{last}(P')$, or, if P' is an if program whose consequent and alternative are P_1 and P_2, and $l_s = \mathit{first}(P')$ and l_f is either $\mathit{first}(P_1)$ or $\mathit{first}(P_2)$. Moreover, we say that l_f is *deeper* that l_s (or, alternatively, that l_s is *shallower* than l_f), denoted $l_s \circledvee l_f$, if either there exists a `while` program fragment P' of P whose body is P'' such that $l_s = \mathit{first}(P')$ and $l_f = \mathit{first}(P'')$, or if there exists a label $l \in \mathit{labels}(P)$ such that $l_s \circledvee l$ and $l \circledcirc l_f$.

For instance, in the program given in Figure 7.2, the following relations hold: $2 \circledcirc 4$, $2 \circledcirc 8$, $4 \circledvee 5$, $4 \circledvee 7$, $1 \circledvee 8$. The following relations do not hold: $3 \circledcirc 5$, $3 \circledvee 5$, $1 \circledvee 5$.

Given a trace θ and a set of trace segments $t_1, t_2 \cdots, t_k$ such that $\theta = t_1 t_2 \cdots t_k$, we say that the sequence $t_1 t_2 \cdots t_k$ is a segmentation of θ.

7.4 Definition Consider a labeled program P and a set of start environments Σ_0, and let θ be a trace of P starting from Σ_0. The *progressive segmentation* of the trace θ is a segmentation $\theta = t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_k\langle l_k, \sigma_k\rangle$, such that $l_i \circledcirc l_{i+1}$ for all i, $1 \leq i < k$. $\square$

Figure 7.2 represents the trace of a program as a sequence of environments attached to program points. Figure 7.1 shows the same trace, represented as a sequence of states on different levels of nesting. The program given in Figure 7.2 has three levels of nesting, the one outside the while loops, the one inside the outer `while` loop, and the one inside the inner loop. These three levels of nesting are represented by horizontal dotted lines in Figure 7.1. Each state is placed on its corresponding line, in order to emphasize the nesting relations between the labels of states in the trace. In order to produce a progressive segmentation of this trace we shall start by identifying the segment $t_3 s_3$ as the segment at the end of the trace made up of states on a level not higher than the level of s_3 (the last state in the trace). To continue, we consider the segment $t_3 s_3$ removed, and we repeat the process until there are no more states left in the trace. That is, we identify the segment $t_2 s_2$ as the segment made up of states on a level not higher than the level of s_2, and after removing this segment, we identify the segment $t_1 s_1$ as the segment made up of states on a level not higher than the level of s_1 (note that t_1 is empty). Given now this segmentations, we need to identify the progressive pairs; they are pairs of consecutive states in the trace that represent a going-around-the-loop transition on the highest level of nesting for that particular segment. For instance, transitions from program point 7 to program point 5 are progressive in segment t_3, but are not progressive in segment t_2. On the other hand, transitions from program point 8 to program point 2 are progressive in segment t_2.

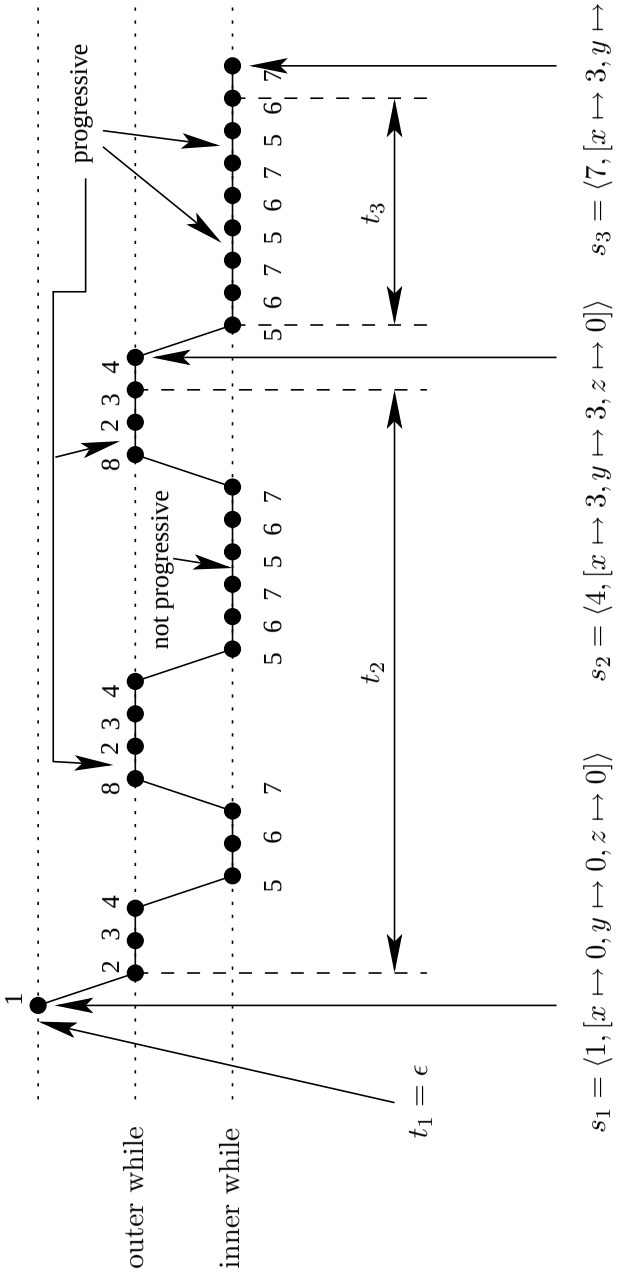


Figure 7.1: Progressive segmentation of a trace

We remark the following: assume θ is a trace of a program P starting from a set of start environments Σ_0, and let $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$ be a segmentation of θ. Denote by θ' the sequence $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$. If $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$ is a progressive segmentation for θ, then $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$ is a progressive segmentation for θ'.

7.3 Uniqueness

Obviously, every trace has at least one progressive segmentation. The following proposition proves that the progressive segmentation of a trace is unique.

7.5 Proposition Given a program P, every trace in $\overrightarrow{P}$ has a unique progressive segmentation.

Proof: We prove the proposition by induction on the number k of segments in the progressive segmentation. That is, we prove that if a trace has a progressive segmentation with k segments, that progressive segmentation is unique.

For the base case, let $k = 1$ and assume that the trace θ has a progressive segmentation $t_1\langle l_1, \sigma_1\rangle$. We prove the base case by reductio ad absurdum and we assume that the progressive segmentation is not unique. Then, there must exist another progressive segmentation $t'_1\langle l'_1, \sigma'_1\rangle \cdots t'_p\langle l'_p, \sigma'_p\rangle$, where $p \geq 1$. Obviously, $l_1 = l'_p$, and $t_1\langle l_1, \sigma_1\rangle = t'_1\langle l'_1, \sigma'_1\rangle \cdots t'_p\langle l'_p, \sigma'_p\rangle$, and therefore l'_{p-1} occurs in t_1. Now, from the fact that $t'_1\langle l'_1, \sigma'_1\rangle \cdots t'_p\langle l'_p, \sigma'_p\rangle$ is a progressive segmentation, it follows that $l'_{p-1} \circledcirc l'_p = l_1$. However, from l'_{p-1} occurs in t_1, it follows that $l'_{p-1} \circledcirc l_1$ cannot be true. Contradiction.

For the induction case, assume that the trace θ has a progressive segmentation $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$, and that this progressive seg-

mentation is not unique. Then, there exists another progressive segmentation $t'_1\langle l'_1, \sigma'_1\rangle \cdots t'_p\langle l'_p, \sigma'_p\rangle$ of θ, where $p \geq 1$. First we prove that $t'_p = t_k$. Assume, by way of contradiction, that this does not happen. From Definition 7.4 we have $l'_{p-1} \otimes l'_p = l_k$. Without loss of generality, assume that t'_p is a proper sub-segment of t_k (the case when t_k is a sub-segment of t'_p is proved in the same way). Then, it must be the case that l'_{p-1} occurs in t_k, which contradicts the fact that $l'_{p-1} \otimes l_k$. Therefore, it must be the case that $t'_p = t_k$. It follows that $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle = t'_1\langle l'_1, \sigma'_1\rangle \cdots t'_{p-1}\langle l'_{p-1}, \sigma'_{p-1}\rangle$. But since $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$ has a progressive segmentation with $k-1$ segments, it follows from the inductive hypothesis that this progressive segmentation is unique. Since both $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$ and $t_k\langle l_k, \sigma_k\rangle$ are unique, it follows that the entire progressive segmentation $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$ is unique. □

7.4 Progressive Index

Given a program P, consider a trace θ and its progressive segmentation $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$. A *progressive pair* is a sequence of two consecutive states $\langle l', \sigma'\rangle\langle l'', \sigma''\rangle$ occurring in a segment $t_i\langle l_i, \sigma_i\rangle$, $1 \leq i \leq k$, such that $l'' \oslash l'$ and $l'' \oslash l_i$.

7.6 Definition Given a program P, let θ be a trace of P, and $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$ its progressive segmentation. The *progressive index* of the trace θ is an index $\tilde{\mu} = \mu_2\mu_3 \cdots \mu_k$, where μ_i is the number of progressive pairs in $t_i\langle l_i, \sigma_i\rangle$, for all i, $2 \leq i \leq k$. □

We note that in the case when $k = 1$ in the definition above, the progressive

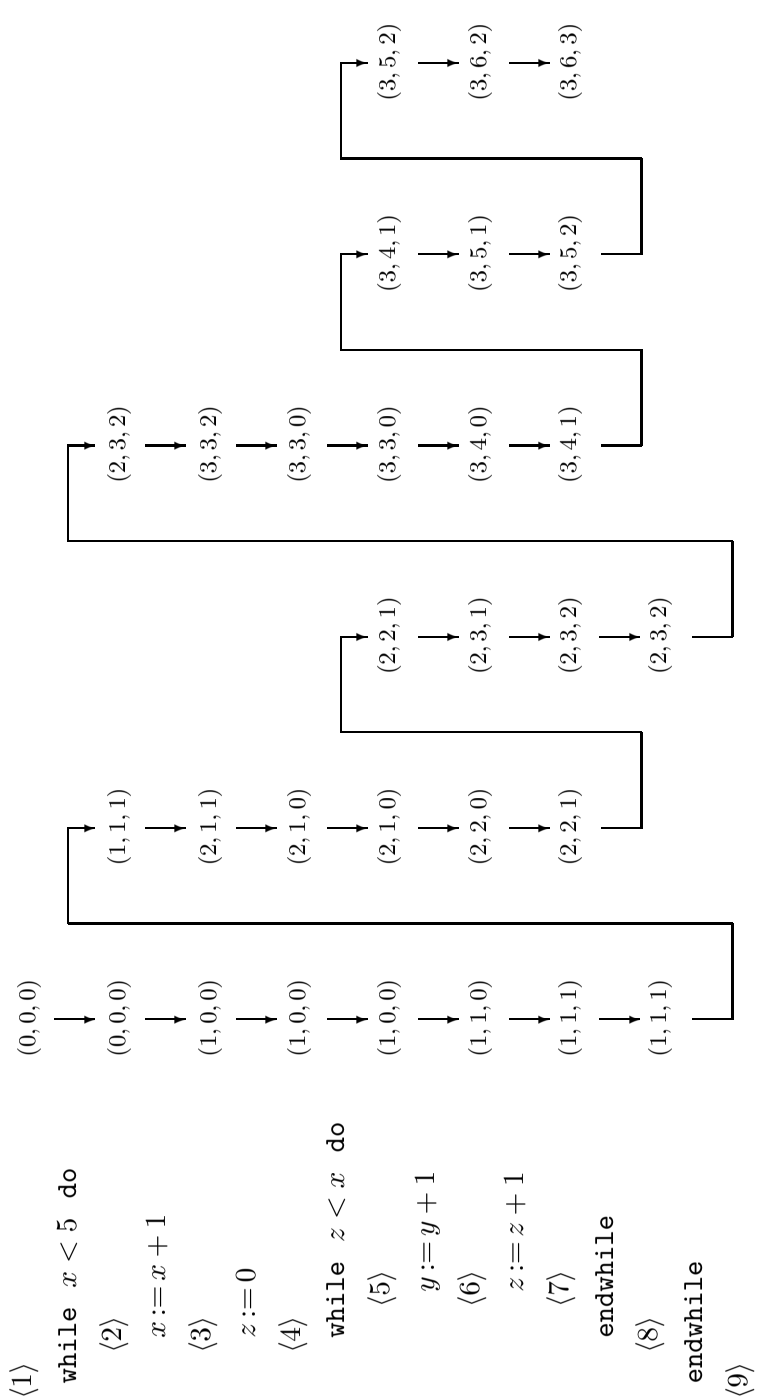


Figure 7.2: Progressive Index

index of the trace θ is ϵ.

The following proposition shows the relation between progressive indices and the length of the computation.

7.7 Proposition Given a program P, let $\theta_1 = t_1\langle l, \sigma_1\rangle$ and $\theta_2 = t_2\langle l, \sigma_2\rangle$ be two traces of P ending at the same program point. Assume that θ_1 is a proper prefix of θ_2, and denote by $\tilde{\mu}_1$ and $\tilde{\mu}_2$ the progressive indices of θ_1 and θ_2, respectively. Then, $\tilde{\mu}_1 < \tilde{\mu}_2$.

Proof: Relegated to Appendix A, on page 276.

The following four propositions establish a link between the syntax of a program and the progressive segmentation of a trace of that program, and shall be useful in defining a configuration based semantic transformer and proving that the semantic transformer is well defined.

7.8 Proposition The following three statements hold.

a) Given a sequence program $P = P_1 \,;\, P_2$ and a trace $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{t_2}_{P_2}$, the progressive index of θ w.r.t. P is the same as the progressive index of t_2 w.r.t. P_2.

b) Given an `if` program P, let P' be either its consequent or its alternative. Consider a trace $\theta \sim \langle \mathit{first}(P), \sigma\rangle \xrightarrow{P} \underbrace{t}_{P'}$. Then, the progressive index of θ w.r.t. P is the same as the progressive index of t w.r.t. P'.

c) Consider a `while` program P whose body is P'. Consider a trace $\theta \sim \langle \mathit{first}(P), \sigma\rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P'} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P'} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P'}$, for some $k > 0$. Denote by $\tilde{\mu}$ the progressive index of t_k w.r.t. P'. Then, the progressive index of θ w.r.t. P is $(k-1)\tilde{\mu}$.

Proof: Relegated to Appendix A, on page 277.

7.9 Proposition Let $P = P_1 \fatsemi P_2$ be a sequence program and $\theta \in \vec{P}$ a trace. The following two statements are true.

a) If $\theta \sim \underbrace{t_1}_{P_1}$ and $rep_{P_1}(t_1) = \vec{K}_1$ for some singleton configuration $\vec{K}_1$ such that $|\vec{K}_1| = P_1$, then $rep_P(\theta) = \vec{K}_1 \fatsemi \vec{K}_{P_2,\perp}$.

b) If $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \langle last(P_1), \sigma\rangle \xrightarrow{P_2} \underbrace{t_2}_{P_2}$ such that $rep_{P_1}(t_1\langle last(P_1), \sigma\rangle) = \vec{K}_1$ and $rep_{P_2}(\langle last(P_1), \sigma\rangle t_2) = \vec{K}_2$ for some singleton configurations $\vec{K}_1$ and $\vec{K}_2$ with $|\vec{K}_1| = P_1$ and $|\vec{K}_2| = P_2$, then $rep_P(\theta) = \vec{K}_1 \fatsemi \vec{K}_2$.

Proof: Relegated to Appendix A, on page 279.

7.10 Proposition Let P be an `if` program whose condition is C and consequent and alternative are P_c and P_a, respectively. Given a trace $\theta \in \vec{P}$, the following statements are true.

a) If $\theta = \langle first(P), \sigma\rangle$ for some environment $\sigma \in \mathbf{Env}$, then $rep_P(\theta) =$ $\langle first(P), \lambda\langle\rangle \,.\, \{\sigma\}\rangle$ `if` C `then` $\vec{K}_{P_c,\perp}$ `else` $\vec{K}_{P_a,\perp}$ `endif` $\langle last(P), \lambda\langle\rangle \,.\, \emptyset\rangle$.

b) If $\theta \sim \langle first(P), \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models C$, and trace segment $t \in \vec{P_c}$, and if $rep_{P_c}(t) = \vec{K}_c$, where $\vec{K}_c$ is a configuration such that $|\vec{K}_c| = P_c$, then $rep_P(\theta) =$ $\langle first(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_c$ `else` $\vec{K}_{P_a,\perp}$ `endif` $\langle last(P), \lambda\langle\rangle \,.\, \emptyset\rangle$.

c) If $\theta \sim \langle first(P), \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} \langle last(P), \sigma_f\rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models C$, and $t \in \vec{P_c}$, and if $rep_{P_c}(t) = \vec{K}_c$, where $\vec{K}_c$ is a configuration such that $|\vec{K}_c| = P_c$, then $rep_P(\theta) =$ $\langle first(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_c$ `else` $\vec{K}_{P_a,\perp}$ `endif` $\langle last(P), \lambda\langle\rangle \,.\, \{\sigma_f\}\rangle$.

d) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models \neg C$, and trace segment $t \in \vec{P_a}$, and if $\mathit{rep}_{P_a}(t) = \vec{K}_a$, where $\vec{K}_a$ is a configuration such that $|\vec{K}_a| = P_a$, then $\mathit{rep}_P(\theta) =$ $\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_{P_c,\perp}$ `else` $\vec{K}_a$ `endif` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset\rangle$.

e) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models \neg C$, and $t \in \vec{P_a}$, and if $\mathit{rep}_{P_a}(t) = \vec{K}_a$, where $\vec{K}_a$ is a configuration such that $|\vec{K}_a| = P_a$, then $\mathit{rep}_P(\theta)\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_{P_c,\perp}$ `else` $\vec{K}_a$ `endif` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\}\rangle$.

Proof: Relegated to Appendix A, on page 280.

We now introduce the operator *seq* for progressive configurations, as a means to combine configurations representing traces of the body of a loop into a single configuration. First we define the operator for annotations, and then we extend it to general progressive configurations.

Formally, given a (possibly infinite) sequence of indexed sets $\Psi_1, \Psi_2, \ldots, \Psi_n, \ldots$, we denote by $\mathit{seq}(\Psi_1, \Psi_2, \ldots, \Psi_n, \ldots)$ the indexed set Ψ defined by

$$\Psi(\tilde{\mu}) = \begin{cases} \emptyset, & \text{if } \tilde{\mu} = \epsilon \\ \Psi_i(\tilde{\mu}') & \text{if } \tilde{\mu} = i\tilde{\mu}' \end{cases}$$

Also, given an indexed set Ψ, and a natural number i, we denote by $\mathit{extr}(\Psi, i)$ the indexed set Ψ' such that $\Psi'(\tilde{\mu}) = \Psi(i\tilde{\mu})$, for all $\mu \in \mathbf{Idx}$. The operators *seq* and *extr* can be extended to progressive configurations in the following way. Given a (possibly infinite) sequence of progressive configurations $K_1, K_2, \ldots, K_n, \ldots$, such that $|K_1| = |K_2| = \cdots = |K_n| = \cdots$, we denote by $\mathit{seq}(K_1, K_2, \ldots, K_n, \ldots)$ the progressive configuration K such that $|K| = |K_1|$ and $K|_l = \mathit{seq}(K_1|_l, K_2|_l, \ldots, K_n|_l, \ldots)$, for all labels $l \in \mathit{labels}(K_1)$. Similarly, given a progressive configuration K, and a

natural number i, we denote by $\mathit{extr}(K, i)$ the progressive configuration K', such that $|K'| = |K|$ and $K'|_l = \mathit{extr}(K|_l, i)$, for all labels $l \in \mathit{labels}(K)$.

7.11 Proposition Let P be a `while` program whose condition and body are C and P_b, respectively. Given a trace $\theta \in \vec{P}$, the following statements are true.

a) If $\theta = \langle \mathit{first}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$ then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma\} \rangle \texttt{ while } C \texttt{ do } \vec{K}_{P_b,\perp} \texttt{ endwhile } \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset \rangle$.

b) If $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\curvearrowright} \langle \mathit{last}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$, such that $\sigma \models \neg C$ then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma\} \rangle \texttt{ while } C \texttt{ do } \vec{K}_{P_b,\perp} \texttt{ endwhile } \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma\} \rangle$.

c) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b}$, for some $k \geq 0$ and some environment $\sigma \in \mathbf{Env}$ such that $\sigma \models C$, and if $\mathit{rep}_{P_b}(t_i) = \vec{K}_i$, $1 \leq i \leq k$, where $\vec{K}_i$ is a configuration such that $|\vec{K}_i| = P_c$, then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \texttt{ while } C \texttt{ do } \mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k) \texttt{ endwhile } \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset \rangle$.

d) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, for some $k \geq 0$ and some environments $\sigma_s, \sigma_f \in \mathbf{Env}$ such that $\sigma_s \models C$ and $\sigma_f \models \neg C$, and if $\mathit{rep}_{P_b}(t_i) = \vec{K}_i$, $1 \leq i \leq k$, where $\vec{K}_i$ is a configuration such that $|\vec{K}_i| = P_c$, then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \texttt{ while } C \texttt{ do } \mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k) \texttt{ endwhile } \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\} \rangle$.

Proof: Relegated to Appendix A, on page 283.

7.5 Representing Traces

We can now represent traces as singleton configurations.

7.12 Definition Consider a trace θ of a program P starting at a set of start environments Σ_0. Let $\vec{K}$ be singleton configuration with $|\vec{K}| = P$, whose annotations are given by:

$$\vec{K}|_l(\tilde{\mu}) = \begin{cases} \{\sigma\}\ , & \text{if there exists a prefix } \langle l_0, \sigma_0\rangle \cdots \langle l, \sigma\rangle \\ & \text{of } \theta \text{ whose progressive index is } \tilde{\mu} \\ \emptyset\ , & \text{otherwise} \end{cases}$$

We say that $\vec{K}$ *represents* the trace θ, and we denote this by $rep_P(\theta) = \vec{K}$. □

According to Proposition 7.7, configuration K above is well defined. Indeed, given a prefix θ' of θ ending at program point l, if $\tilde{\mu}$ is the progressive index of θ', then the mapping $\theta' \mapsto (l, \tilde{\mu})$ is injective. This justifies the fact that $\vec{K}|_l(\tilde{\mu})$ is either a singleton or an empty set. In what follows, we shall prove that the representation relation is well defined. The first step is proving that the relation defines in fact an injective mapping.

7.13 Proposition Given a program P, the mapping $\theta \mapsto rep_P(\theta)$ is injective.

Proof: Assume that the mapping is not injective. Then there are two traces $\theta_1, \theta_2 \in \vec{P}$ that are mapped into the same configuration $\vec{K}$. Let $l = \mathit{first}(\vec{K})$ and assume that $\vec{K}|_l = \{\sigma\}$. Then, $\langle l, \sigma\rangle$ is the start state of both θ_1 and θ_2. According to Proposition 5.4, either θ_1 is a prefix of θ_2 or θ_2 is a prefix of θ_1. Without loss of generality, we may assume that θ_1 is a prefix of θ_2. Let now $\langle l', \sigma'\rangle$ be the state following θ_1 in θ_2, and denote by $\tilde{\mu}$ the progressive index of trace $\theta_1\langle l', \sigma'\rangle$. Since $\langle l', \sigma'\rangle$ does not occur in θ_1, we have $\vec{K}|_{l'}(\tilde{\mu}) = \emptyset$. On the other hand, $\langle l', \sigma'\rangle$ occurs in θ_2, and therefore $\vec{K}|_{l'}(\tilde{\mu}) = \{\sigma'\}$. Contradiction. □

$$\textit{assign} : \mathbf{Var} \times \mathbf{Expr} \times \mathbf{Idx} \mapsto \mathbf{Idx}$$

$$\textit{assign}(x, E, \Psi) \quad = \quad \lambda\tilde{\mu}\,.\,\{\sigma \,|\, \text{exists } \sigma' \in \Psi(\tilde{\mu}) \text{ s.t. } \sigma = \sigma'[x \mapsto E(\sigma)]\}$$

$$\textit{filter} : \mathbf{Constr} \times \mathbf{Idx} \mapsto \mathbf{Idx}$$

$$\textit{filter}(C, \Psi) \quad = \quad \lambda\tilde{\mu}\,.\,\{\sigma \,|\, \sigma \in \Psi(\tilde{\mu}) \text{ and } \sigma \models C\}$$

$$\cup : \mathbf{Idx} \times \mathbf{Idx} \mapsto \mathbf{Idx}$$

$$\Psi_1 \cup \Psi_2 \quad = \quad \lambda\tilde{\mu}\,.\,\Psi_1(\tilde{\mu}) \cup \Psi_2(\tilde{\mu})$$

$$\textit{before} : \mathbf{Idx} \times \mathbf{Idx} \mapsto \mathbf{Idx}$$

$$\textit{before}(\Psi_1, \Psi_2) \quad = \quad \lambda\tilde{\mu}\,.\,\begin{cases} \emptyset\ , & \text{if } \tilde{\mu} = \epsilon \\ \Psi_1(\tilde{\mu}')\ , & \text{if } \tilde{\mu} = \tilde{\mu}'0 \\ \Psi_2(\tilde{\mu}'(\mu - 1))\ , & \text{if } \tilde{\mu} = \tilde{\mu}'\mu \text{ and } \mu > 0 \end{cases}$$

$$\textit{collect} : \mathbf{Idx} \mapsto \mathbf{Idx}$$

$$\textit{collect}(\Psi) \quad = \quad \lambda\tilde{\mu}\,.\,\bigcup_{\mu \geq 0} \Psi(\tilde{\mu}\mu)$$

Figure 7.3: Indexed Set Operators

7.6 Transition Relation for Singleton Configurations

Figure 7.3 introduces several operators for indexed sets. These operators shall be useful in defining a semantic transformer for the configuration-based trace semantics, as well as the progressive semantics.

The operator *assign* models the effect of assignment statements in the progressive

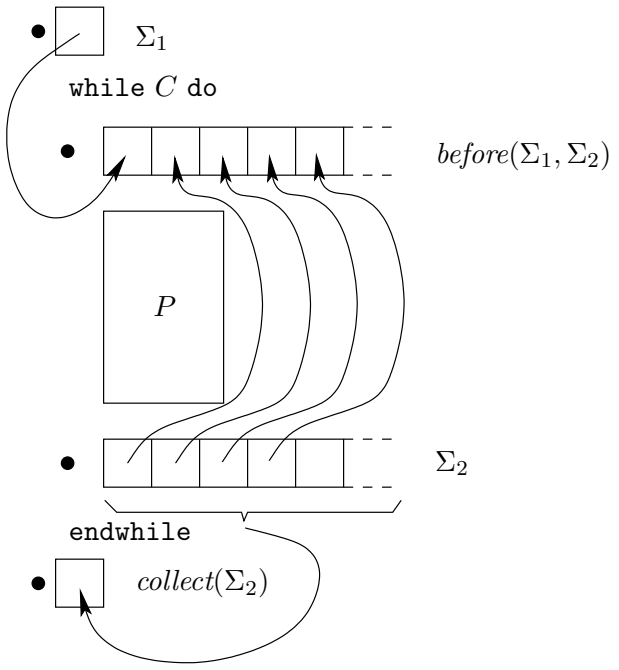


Figure 7.4: The *before* and *collect* operators explained

semantics. The operator *filter* models the effect of branching in `if` and `while` statements. We also overload the union operator $\cup$ to model the effect of two `if` branches being joined at the end of an `if` statement. The operators *before* and *collect* are essential in capturing the sequences of environments that occur at program points inside `while` loops. Figure 7.4 explains how these operators work. The *before* operator builds the sequence of environments at the first program point of the body of a `while` loop. The first environment in the sequence comes from the program point outside the loop, while all the subsequent environments come from the bottom of the loop. The *collect* operator builds the sequence of states that are transferred from the last program point of the body of a `while` loop to the program point outside the `while` loop.

The following two remarks show the relationship between the configuration se-

$$\langle l_s, \Psi_s\rangle \,\texttt{skip}\, \langle l_f, \Psi_f\rangle \;\longrightarrow\; \langle l_s, \Psi_s\rangle \,\texttt{skip}\, \langle l_f, \Psi_s\rangle$$

$$\langle l_s, \Psi_s\rangle\, x := E\, \langle l_f, \Psi_f\rangle \;\longrightarrow\; \langle l_s, \Psi_s\rangle\, x := E\, \langle l_f, \mathit{assign}(x, E, \Psi_s)\rangle$$

$$\begin{array}{l} \langle l_s, \Psi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \quad\quad K_1 \\ \quad \texttt{else} \\ \quad\quad K_2 \\ \texttt{endif} \\ \langle l_f, \Psi_f\rangle \end{array} \longrightarrow \begin{array}{l} \langle l_s, \Psi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \quad\quad \langle \mathit{first}(K_1), \mathit{filter}(C, \Psi_s)\rangle \fatsemi K_1' \\ \quad \texttt{else} \\ \quad\quad \langle \mathit{first}(K_2), \mathit{filter}(\neg C, \Psi_s)\rangle \fatsemi K_2' \\ \texttt{endif} \\ \langle l_f, K_1|_{\mathit{last}(K_1)} \cup K_2|_{\mathit{last}(K_2)}\rangle \end{array}$$

where $K_1 \longrightarrow K_1'$ and $K_2 \longrightarrow K_2'$

$$\begin{array}{l} \langle l_s, \Psi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad K \\ \texttt{endwhile} \\ \langle l_f, \Psi_f\rangle \end{array} \longrightarrow \begin{array}{l} \langle l_s, \Psi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \langle \mathit{first}(K), \Psi'\rangle \fatsemi K' \\ \texttt{endwhile} \\ \langle l_f, \Psi''\rangle \end{array}$$

where $K \longrightarrow K'$
$\Psi' = \mathit{filter}(C, \mathit{before}(\Psi_s, K|_{\mathit{last}(K)}))$
$\Psi'' = \mathit{filter}(\neg C, \Psi_s \cup \mathit{collect}(K|_{\mathit{last}(K)}))$

$$K_1 \fatsemi K_2 \;\longrightarrow\; K_1' \fatsemi K_2'$$

where $K_1 \longrightarrow K_1'$
$K_2 \longrightarrow K_2'$

Figure 7.5: Transition Relation for Configurations

quencing operator *seq*, the operators in Figure 7.4 and the syntax of our programming language.

7.14 Remark The sequencing operator distributes over the operators defined in Figure 7.3. Indeed, we have

a) $\mathit{seq}(\mathit{before}(\Psi_{11}, \Psi_{12}), \ldots, \mathit{before}(\Psi_{k1}, \Psi_{k2})) =$

$\mathit{before}(\mathit{seq}(\Psi_{11}, \ldots, \Psi_{k1}), \mathit{seq}(\Psi_{12}, \ldots, \Psi_{k2}))$, $k > 0$;

b) $seq(assign(x, E, \Psi_1), \ldots, assign(x, E, \Psi_k)) = assign(x, E, seq(\Psi_1, \ldots, \Psi_k))$;

c) $seq(filter(C, \Psi_1), \ldots, filter(C, \Psi_k)) = filter(C, seq(\Psi_1, \ldots, \Psi_k))$;

d) $seq(\Psi_{11} \cup \Psi 12, \ldots, \Psi_{k1} \cup \Psi_{k2}) = seq(\Psi_{11}, \ldots, \Psi_{k1}) \cup seq(\Psi 12, \ldots, \Psi_{k2})$;

e) $seq(collect(\Psi_1), \ldots, collect(\Psi_k)) = collect(seq(\Psi_1, \ldots, \Psi_k))$.

□

7.15 Remark The *seq* operator distributes over skip statements and assignments, as well as over the sequence, if, and while constructs.

a) Let $P = \langle l_s \rangle \, \texttt{skip} \, \langle l_f \rangle$ be a skip program, and $K_1, \ldots, K_n$ a set of configurations such that $|K_i| = P$, for all i, $1 \leq i \leq n$. Each K_i can be written as $\langle l_s, \Sigma_i \rangle \, \texttt{skip} \, \langle l_f, \Sigma_i \rangle$. Then,
$seq(K_1, \ldots, K_n) = seq(\langle l_s, \Sigma_1 \rangle, \ldots, \langle l_s, \Sigma_n \rangle) \, \texttt{skip} \, seq(\langle l_f, \Sigma_1 \rangle, \ldots, \langle l_f, \Sigma_n \rangle)$.

b) Let $P = \langle l_s \rangle \, x := E \, \langle l_f \rangle$ be an assignment, and $K_1, \ldots, K_n$ a set of configurations such that $|K_i| = P$, for all i, $1 \leq i \leq n$. Each K_i can be written as $\langle l_s, \Sigma_i \rangle \, x := E \, \langle l_f, \Sigma_i' \rangle$, where $\Sigma_i' = assign(x, E, \Sigma_i)$. Then,
$seq(K_1, \ldots, K_n) = seq(\langle l_s, \Sigma_1 \rangle, \ldots, \langle l_s, \Sigma_n \rangle) \, x :=$
$E \, seq(\langle l_f, \Sigma_1' \rangle, \ldots, \langle l_f, \Sigma_n' \rangle)$.

c) Let $P = P_1 \, ; \, P_2$ be a sequence program, and $K_1, \ldots, K_n$ a set of configurations such that $|K_i| = P$, for all i, $1 \leq i \leq n$. Each K_i can be written as $K_i' \, ; \, K_i''$. Then,
$seq(K_1, \ldots, K_n) = seq(K_1', \ldots, K_n') \, ; \, seq(K_1'', \ldots, K_n'')$.

d) Let $P = \langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$ be an `if` program and

$K_1, \ldots, K_n$ a set of configurations such that $|K_i| = P$, for all i, $1 \leq i \leq n$.

Each K_i can be written as $\langle l_s, \Sigma_{si} \rangle$ `if` C `then` K_{ci} `else` K_{ai} `endif` $\langle l_f, \Sigma_{fi} \rangle$.

Then, $seq(K_1, \ldots, K_n) = seq(\langle l_s, \Sigma_{s1} \rangle, \ldots, \langle l_s, \Sigma_{sn} \rangle)$ `if` C `then`

$seq(K_{c1}, \ldots, K_{cn})$ `else` $seq(K_{a1}, \ldots, K_{an})$ `endif`

$seq(\langle l_f, \Sigma_{f1} \rangle, \ldots, \langle l_f, \Sigma_{fn} \rangle)$.

e) Let $P = \langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$ be a `while` program and

$K_1, \ldots, K_n$ a set of configurations such that $|K_i| = P$, for all i, $1 \leq i \leq n$.

Each K_i can be written as $\langle l_s, \Sigma_{si} \rangle$ `while` C `do` K_{bi} `endwhile` $\langle l_f, \Sigma_{fi} \rangle$.

Then, $seq(K_1, \ldots, K_n) = seq(\langle l_s, \Sigma_{s1} \rangle, \ldots, \langle l_s, \Sigma_{sn} \rangle)$ `while` C `do`

$seq(K_{b1}, \ldots, K_{bn})$ `endwhile` $seq(\langle l_f, \Sigma_{f1} \rangle, \ldots, \langle l_f, \Sigma_{fn} \rangle)$.

□

Singleton configurations capture the history of the program's execution in a syntax-based manner. With every transition performed during the execution of the program, a new environment must be added to the singleton configuration that captures the execution's history. This process can be modeled using the transition relation for singleton and progressive configurations given in Figure 7.5.

7.7 Correctness

In what follows, we shall prove that the transition relation introduced above is well defined, that is, given two traces θ_1 and θ_2 of a program P such that θ_1 is the longest proper prefix of θ_2, we have $rep_P(\theta_1) \longrightarrow rep_P(\theta_2)$. We start with the following two propositions which prove several simple properties of the transition relation.

7.16 Proposition Let P be a program and $\vec{K}_1, \ldots, \vec{K}_k$ and $\vec{K}'_1, \ldots, \vec{K}'_k$ two sets of singleton configurations such that $|\vec{K}_i| = |\vec{K}'_i| = P$, for all i, $1 \leq i \leq k$. Assume

that $\vec{K}_i \longrightarrow \vec{K}'_i$. Then, $seq(\vec{K}_1, \ldots, \vec{K}_i) \longrightarrow seq(\vec{K}'_1, \ldots, \vec{K}'_i)$, $k > 0$.

Proof: Relegated to Appendix A, on page 286.

7.17 Proposition Let P be a program. The following two statements hold.

a) Denote by $\vec{K}_{P,\perp}$ the empty configuration corresponding to P. Then, $\vec{K}_{P,\perp} \longrightarrow \vec{K}_{P,\perp}$.

b) Let θ be a terminal trace w.r.t P, and denote by $\vec{K}$ the configuration such that $rep_P(\theta) = \vec{K}$. Then $K \longrightarrow K$.

Proof: Relegated to Appendix A, on page 288.

The following proposition shows that the transition relation defined in Figure 7.5 is well-defined.

7.18 Proposition Consider a labeled program P and let $\theta_1, \theta_2 \in \overrightarrow{P}$ be two traces such that θ_1 is the longest proper prefix of θ_2. Let $\vec{K}_1$ and $\vec{K}_2$ be two singleton configurations such that $rep_P(\theta_1) = \vec{K}_1$ and $rep_P(\theta_2) = \vec{K}_2$. Then, $\vec{K}_1 \longrightarrow \vec{K}_2$.

Proof: Relegated to Appendix A, on page 291.

Proposition 7.18 shows that Equation 7.12 defines a sound representation of traces as configurations. In what follows, we will consistently use singleton configurations instead of traces. As a result, the trace semantics becomes a set of singleton configurations.

Chapter 8

Sets of States as Configurations

In the previous chapter we have shown that program traces can be expressed as singleton configurations, and argued that singleton configurations are a compositional, syntax directed means of representing program behavior. In what follows, we shall continue our quest of making configurations a pervasive and uniform means of expressing program semantics. In this chapter, we show that sets of states can be expressed as configurations, and we re-express some of the collecting-semantics-related results in a configuration-based setting.

8.1 Collecting Configurations

Given a program P, let S be a set of states of P. The *collecting configuration that represents* S is a configuration $\overline{K}$ such that $|\overline{K}| = P$ and $\overline{K}|_l = \{\sigma \mid \langle l, \sigma \rangle \in S\}$. We note that if $\overline{K'}$ is a sub-configuration of $\overline{K}$, then $\overline{K'}$ represents the subset of S whose states have labels in $\mathit{labels}(\overline{K'})$. We also note that if S_1 and S_2 are two sets of states represented by K_1 and K_2, respectively, then $S_1 \subseteq S_2$ is equivalent to $K_1 \preccurlyeq K_2$ We also extend the union operator to collective configurations in the following way. Given two collective configurations $\overline{K}_1$ and $\overline{K}_2$ such that $|\overline{K}_1| = |\overline{K}_2|$, the union

$\langle 1, \top \rangle$
  $x := 0$
$\langle 2, \{[x \mapsto 0, y \mapsto 0], [x \mapsto 0, y \mapsto -1], [x \mapsto 0, y \mapsto 1], \cdots\} \rangle$
  $y := 0$
$\langle 3, \{[x \mapsto 0, y \mapsto 0]\} \rangle$
  `while` $x < 10$ `do`
    $\langle 4, \{[x \mapsto 0, y \mapsto 0], [x \mapsto 1, y \mapsto 1], \cdots, [x \mapsto 9, y \mapsto 45]\} \rangle$
      $x := x + 1$
    $\langle 5, \{[x \mapsto 1, y \mapsto 0], [x \mapsto 2, y \mapsto 1], \cdots, [x \mapsto 10, y \mapsto 45]\} \rangle$
      $y := y + x$
    $\langle 6, \{[x \mapsto 1, y \mapsto 1], [x \mapsto 2, y \mapsto 3], \cdots, [x \mapsto 10, y \mapsto 55]\} \rangle$
  `endwhile`
$\langle 7, \{[x \mapsto 10, y \mapsto 55]\} \rangle$

Figure 8.1: Example of Collecting Configuration

$\overline{K}_1 \cup \overline{K}_2$ is defined by $(\overline{K}_1 \cup \overline{K}_2)|_l = \overline{K}_1|_l \cup \overline{K}_2|_l$, for all labels $l \in \mathit{labels}(\overline{K}_1)$. We note now that, given a program P, $(\{\overline{K} \,|\, |\overline{K}| = P\}, \preccurlyeq)$ is a complete lattice.

Figure 8.1 shows an example of a collecting configuration which represents the collecting semantics of its underlying program.

8.1 Proposition The mapping $S \mapsto \overline{K}$, where $\overline{K}$ is the collecting configuration that represents the set of states S, is bijective.

Proof: We will prove that the mapping is both injective and surjective. Assume the mapping were not injective. Then, there would be two distinct sets of states S_1 and S_2 that would be mapped into the same configuration $\overline{K}$. Let $\langle l, \sigma \rangle \in S_1 \backslash S_2$ (if $S_1 \backslash S_2$ is empty, we could choose a state in $S_2 \backslash S_1$ and obtain a similar proof). From $\langle l, \sigma \rangle \in S_1$ we have that $\sigma \in \overline{K}|_l$. From $\langle l, \sigma \rangle \notin S_2$ we have $\sigma \notin \overline{K}|_l$. Contradiction. This proves that the mapping is injective. In order to prove that the mapping is

surjective, we note that given a configuration $\overline{K}$, the set $S = \cup_{l \in labels(\overline{K})} \overline{K}|_l$ is represented by $\overline{K}$, and therefore, for every collecting configuration, there exists a set of states represented by it. □

8.2 Transfer Function

Given a program P, the collecting semantics has been defined as the fixpoint of the semantic transformer T_P. A similar semantic transformer for semantic configurations is defined in Figure 8.2. We note here that $|\overline{K}| = |T(\overline{K})|$, for all collecting configurations $\overline{K}$. The following proposition shows that collecting configurations are a sound way of representing the collecting semantics and that the semantic transformer T is well-defined.

8.2 Proposition Let P be a labeled program, and S a set of states of P. Denote by S_0 the set $\{\langle l, \sigma\rangle \mid \langle l, \sigma\rangle \in S \text{ and } l = \mathit{first}(P)\}$. Assume that the representation of S is a configuration $\overline{K}$. Then $(T_P \cup S_0)(S)$ is represented by $T(\overline{K})$.

Proof: Relegated to Appendix A, on page 292.

8.3 Fixpoint Characterization

Given a program P, according to Proposition 6.2, the set $\overline{\Gamma}_P$ of all collecting configurations whose skeleton is P, together with the induced partial order $\preccurlyeq$, is a complete lattice. It easy to prove that the restriction T' of the operator T to $\overline{\Gamma}_P$ is continuous. From Proposition 2.1, it follows that T' has a least fixpoint in $\overline{\Gamma}_P$. The following proposition shows that the collecting semantics of P is a fixed point of T'.

8.3 Proposition Let P be a program, Σ_0 a set of start environments, and denote by $\overline{K}_0$ the configuration such that $|\overline{K}_0| = P$, $\overline{K}_0|_{\mathit{first}(P)} = \Sigma_0$, and $\overline{K}_0|_l = \emptyset$ for all

$$T\left(\begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{skip}\\ \langle l_f, \Sigma_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{skip}\\ \langle l_f, \Sigma_s\rangle\end{array}$$

$$T\left(\begin{array}{l}\langle l_s, \Sigma_s\rangle\\ x := E\\ \langle l_f, \Sigma_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Sigma_s\rangle\\ x := E\\ \langle l_f, \{\sigma[x \mapsto E] \mid \sigma \in \Sigma_s\}\rangle\end{array}$$

$$T\left(\begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{if } C\\ \quad \texttt{then}\\ \quad\quad \overline{K}_c\\ \quad \texttt{else}\\ \quad\quad \overline{K}_a\\ \texttt{endif}\\ \langle l_f, \Sigma_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{if } C\\ \quad \texttt{then}\\ \quad\quad \langle \mathit{first}(\overline{K}_c), \Sigma_s \cap \{\sigma \mid \sigma \models C\}\rangle \fatsemi T(\overline{K}_c)\\ \quad \texttt{else}\\ \quad\quad \langle \mathit{first}(\overline{K}_a), \Sigma_s \cap \{\sigma \mid \sigma \models \neg C\}\rangle \fatsemi T(\overline{K}_a)\\ \texttt{endif}\\ \langle l_f, \overline{K}_c|_{\mathit{last}(\overline{K}_c)} \cup \overline{K}_a|_{\mathit{last}(\overline{K}_a)}\rangle\end{array}$$

$$T\left(\begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{while } C \texttt{ do}\\ \quad \overline{K}_b\\ \texttt{endwhile}\\ \langle l_f, \Sigma_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Sigma_s\rangle\\ \texttt{while } C \texttt{ do}\\ \quad \langle \mathit{first}(\overline{K}_b), \Sigma\rangle \fatsemi T(\overline{K}_b)\\ \texttt{endwhile}\\ \langle l_f, \{\sigma \mid \sigma \models C\} \cap (\Sigma_s \cup \overline{K}_b|_{\mathit{last}(\overline{K}_b)})\rangle\end{array}$$

where $\Sigma = \{\sigma \mid \sigma \models C\} \cap (\Sigma_s \cup \overline{K}_b|_{\mathit{last}(\overline{K}_b)})$

$$T\left(\overline{K}_1 \fatsemi \overline{K}_2\right) = T\left(\overline{K}_1\right) \fatsemi T\left(\overline{K}_2\right)$$

Figure 8.2: Transfer Function for Collecting Configurations

labels $l \in \mathit{labels}(P) \backslash \{\mathit{first}(P)\}$. Then, $T^{\omega}(\overline{K}_0)$ represents the collecting semantics of P w.r.t. Σ_0.

Proof: Relegated to Appendix A, on page 295.

8.4 Properties

From Proposition 8.3 and Remark 2.5 it follows that the collecting semantics of a program P is $\mathit{lfp}(T' \cup \overline{K}_0)$, where $\overline{K}_0$ is the start configuration, as defined in Proposition 8.3.

The following proposition shows that the transfer operator T distributes over a union of collective configurations and will be useful in establishing a relationship between the configuration-based collecting semantics and the progressive semantics.

8.4 Proposition Given a set of collective configurations with the same skeleton $\overline{\Gamma}$, we have that $T(\bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}) = \bigcup_{\overline{K} \in \overline{\Gamma}} T(\overline{K})$.

Proof: Relegated to Appendix A, on page 295.

8.5 Discussion

Proposition 8.3 proves that the collecting semantics of a program P w.r.t. a set of start environments Σ_0 is the least fixpoint of the T operator over the lattice $L = (\{\overline{K} \,|\, |\overline{K}| = P \text{ and } \overline{K}|_{\mathit{first}(P)} = \Sigma_0\}, \preccurlyeq)$. Typically, classic program reasoning methods would either compute or verify that a set of states is a superset of the collecting semantics. The fixpoint characterization that we provided induces a very convenient means of performing such verification. Since a postfixpoint of T w.r.t. the lattice given above is necessarily a superset of the least fixpoint, one sufficient condition for a collecting configuration $\overline{K}$ to be a superset of the collecting

$\langle 1, \{\sigma \mid \sigma \in \mathbf{Env}\}\rangle$
  $x := 0$
$\langle 2, \{\sigma \mid \sigma(x) = 0\}\rangle$
  $y := 0$
$\langle 3, \{\sigma \mid \sigma(x) = 0 \text{ and } \sigma(y) = 0\}\rangle$
  `while` $x < 10$ `do`
    $\langle 4, \{\sigma \mid 0 \leq \sigma(x) < 10 \text{ and } \sigma(y) \in \{0, 1, 3, 6, 10, 15, 21, 28, 36, 45\}\}\rangle$
      `if` $x \neq -1$ `then`
        $\langle 5, \{\sigma \mid 0 \leq \sigma(x) < 10 \text{ and } \sigma(y) \in \{0, 1, 3, 6, 10, 15, 21, 28, 36, 45\}\}\rangle$
          $x := x + 1$
        $\langle 6, \{\sigma \mid 1 \leq \sigma(x) < 11 \text{ and } \sigma(y) \in \{0, 1, 3, 6, 10, 15, 21, 28, 36, 45\}\}\rangle$
          $y := y + x$
        $\langle 7, \{\sigma \mid 1 \leq \sigma(x) < 11 \text{ and } \sigma(y) \in \{1, 3, 6, 10, 15, 21, 28, 36, 45, 55\}\}\rangle$
      `else`
        $\langle 8, \emptyset\rangle$
          `skip`
        $\langle 9, \emptyset\rangle$
      `endif`
    $\langle 10, \{\sigma \mid 1 \leq \sigma(x) < 11 \text{ and } \sigma(y) \in \{1, 3, 6, 10, 15, 21, 28, 36, 45, 55\}\}\rangle$
  `endwhile`
$\langle 11, \{\sigma \mid \sigma(x) = 10 \text{ and } \sigma(y) = 55\}\rangle$

Figure 8.3: Collecting Semantics as the Least Fixpoint

semantics is to have $T(\overline{K}) \preccurlyeq \overline{K}$. Once we have found a collective configuration $\overline{K}$ that satisfies this condition, we can take advantage of the fact that the T operator is monotone, and harness it to produce more refined approximations of the collecting semantics. This can be achieved by computing the sequence $(T^k(\overline{K}))_{k\geq 0}$. This sequence is monotonically decreasing, and shall produce increasingly more precise approximations of the collecting semantics. However, since the T operator does not necessarily have a unique fixpoint, the sequence of approximations does not

$\langle 1, \{\sigma \mid \sigma \in \mathbf{Env}\}\rangle$
  $x := 0$
$\langle 2, \{\sigma \mid \sigma(x) = 0\}\rangle$
  $y := 0$
$\langle 3, \{\sigma \mid \sigma(x) = 0 \text{ and } \sigma(y) = 0\}\rangle$
  `while` $x < 10$ `do`
    $\langle 4, \{\sigma \mid \sigma(x) < 10\}\rangle$
      `if` $x \neq -1$ `then`
        $\langle 5, \{\sigma \mid \sigma(x) < 10 \text{ and } \sigma(x) \neq -1\}\rangle$
          $x := x + 1$
        $\langle 6, \{\sigma \mid \sigma(x) < 11 \text{ and } \sigma(x) \neq 0\}\rangle$
          $y := y + x$
        $\langle 7, \{\sigma \mid \sigma(x) < 11 \text{ and } \sigma(x) \neq 0\}\rangle$
      `else`
        $\langle 8, \{\sigma \mid \sigma(x) = -1\}\rangle$
          `skip`
        $\langle 9, \{\sigma \mid \sigma(x) = -1\}\rangle$
      `endif`
    $\langle 10, \{\sigma \mid \sigma(x) < 11 \text{ and } \sigma(x) \neq 0\}\rangle$
  `endwhile`
$\langle 11, \{\sigma \mid \sigma(x) = 10\}\rangle$

Figure 8.4: Greatest Fixpoint

get arbitrarily close to the least fixpoint of T. We illustrate this situation with an example. Figures 8.3 and 8.4 show a collecting semantics of a program, and an approximation $\overline{K}$ of that collecting semantics that happens to be the greatest fixpoint of T on the lattice L. since the configuration given in Figure 8.4 is a fixpoint of T, the sequence $(T^k(\overline{K}))_{k\geq 0}$ will not improve the precision of $\overline{K}$. As it can be seen from Figure 8.3, program points 8 and 9 are not live, while the values of x and y are positive throughout the entire execution of the program fragment. However,

the approximation given in Figure 8.4 shows non-empty sets of environments for program points 8 and 9, and negative integers as possible values for variable x at all program points inside the loop. For this reason, the collecting semantics and T operator cannot be the basis for a reasoning framework that would infer or verify *liveness* properties.

Chapter 9

The Progressive Middle-Ground

In the previous two chapters we have introduced configurations as a uniform, syntax-directed means of expressing semantics, and showed that both the trace and the collecting semantics can be represented as configurations. However, each of these semantics has its own shortcomings. One one hand, the trace semantics, while capturing the behavior of the program in full detail, is too low level to be amenable to program reasoning. On the other hand, collecting semantics abstracts away all the sequencing information, becoming unable to deal with liveness and progress properties. However, collecting semantics brings in a very useful feature, that is, the ability to reason about programs in a compositional manner, with (safety) properties expressed as formulas attached to program points. Such degree of locality fosters the modular development of software and is a feature that we would like to retain in our framework.

What we need is a middle-ground, a semantics whose level of abstraction lies in-between the trace and collecting semantics, such that it allows reasoning about liveness and progress properties, while retaining the very useful feature of having properties about programs expressed as formulas attached to program points. We

shall argue in this chapter that the progressive semantics is such a middle ground, exhibiting a level of abstraction that makes it amenable to reasoning about both liveness and safety properties in a unified compositional framework.

9.1 Progressive Semantics

We start with a configuration-based definition of the progressive semantics.

9.1 Definition The *progression* of an annotated program P and a set of start states Σ_0 is a progressive configuration K such that $|K| = P$ and $K|_l(\tilde{\mu})$ is the set of all environments σ for which there is a trace starting in Σ_0, ending with state $\langle l, \sigma \rangle$ and whose progressive index is $\tilde{\mu}$. □

Figure 9.2 shows a progression of the program introduced in Figure 7.2. Figure 9.3 shows a progression of the same program, using the indexed set language introduced in Section 6.1.

In what follows, we present a fixed point specification of the progressive semantics. As argued in Section 2.2 such a specification relies on a semantic domain which must be a complete lattice, and on a semantic transformer, which must be a monotone mapping over the semantic domain. For a given program P, our semantic domain is the set $\mathbf{Progressive}(P)$ and Proposition 6.2 shows that $(\mathbf{Progressive}(P), \preccurlyeq)$ is a complete lattice. In order to define a semantic transformer, we first need to introduce a set of operators over indexed sets. These operators are defined in Figure 7.3. We define the operator $\vec{T} : \bigcup_P 2^{\mathbf{Singletons}(P)} \mapsto \bigcup_P 2^{\mathbf{Singletons}(P)}$ by $\vec{T}(\Gamma) = \{\vec{K} \mid \text{there exists a configuration } \vec{K}' \in \Gamma \text{ such that } \vec{K}' \longrightarrow \vec{K}\}$.

9.2 Remark From Propositions 5.7 and 7.18 it follows that, given a program P and

a set of start environments Σ_0, the configuration-based trace semantics is $\mathit{lfp}(\vec{T}'\cup\vec{\Gamma}_0)$, where $\vec{T}'$ is the restriction of $\vec{T}$ to the set $2^{\vec{\Gamma}_P}$, where $\vec{\Gamma} = \{\vec{K} \,|\, |\vec{K}| = P\}$, and $\vec{\Gamma}_0$ is the set of singleton configurations $\{\vec{K} \,|\, |\vec{K}| = P,\, \vec{K}_0|_{\mathit{first}(P)} = \lambda\langle\rangle\,.\,\{\sigma\},\, \sigma \in \Sigma_0, \vec{K}_0|_l = \lambda\langle\rangle\,.\,\emptyset$ for all $l \in \mathit{labels}(P)\backslash\{l\}\}$. □

We define the operator T^p : **AProg**(**Labels** × **IdxEnv**) ↦ **AProg**(**Labels** × **IdxEnv**) by $T^p(K_1) = K_2$ if $K_1 \longrightarrow K_2$. Figure 9.1 presents an alternative syntax-based definition of the T^p operator. T^p is monotone.

9.3 Proposition Given a set of progressive configurations with the same skeleton Γ, we have that

$$T^p\left(\bigcup_{K\in\Gamma} K\right) = \bigcup_{K\in\Gamma} T^p(K)\ .$$

Proof: The proof is similar to the one of Proposition 8.4 as the T^p operator has a syntax-directed definition that is similar to T, and the $\cup$ operator for indexed sets is distributive over configurations and other indexed set operators. □

9.2 Fixpoint Characterization

In this section we show that the T^p operator has a unique fixpoint w.r.t. a given program and a set of start environments. It is also true that this fixpoint is the progression of P w.r.t. Σ_0, but this result, given in Theorem 10.5, shall be postponed till Section 10.1. We start with a set of helper propositions that shall be useful in constructing an inductive proof. The main result of this section is given in Theorem 9.7.

$$T^p\left(\begin{array}{l}\langle l_s, \Psi_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \Psi_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Psi_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \Psi_s\rangle\end{array}$$

$$T^p\left(\begin{array}{l}\langle l_s, \Psi_s\rangle \\ \quad x := E \\ \langle l_f, \Psi_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Psi_s\rangle \\ \quad x := E \\ \langle l_f, \mathit{assign}(x, e, \Psi_s)\rangle\end{array}$$

$$T^p\left(\begin{array}{l}\langle l_s, \Psi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \qquad K_1 \\ \quad \texttt{else} \\ \qquad K_2 \\ \texttt{endif} \\ \langle l_f, \Psi_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Psi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \qquad \langle l_{1s}, \mathit{filter}(C, K_1|_{l_{1s}})\rangle \fatsemi T^p(K_1) \\ \quad \texttt{else} \\ \qquad \langle l_{2s}, \mathit{filter}(\neg C, K_2|_{l_{2s}})\rangle \fatsemi T^p(K_2) \\ \texttt{endif} \\ \langle l_f, K_1|_{l_{1f}} \cup K_2|_{l_{2f}}\rangle\end{array}$$

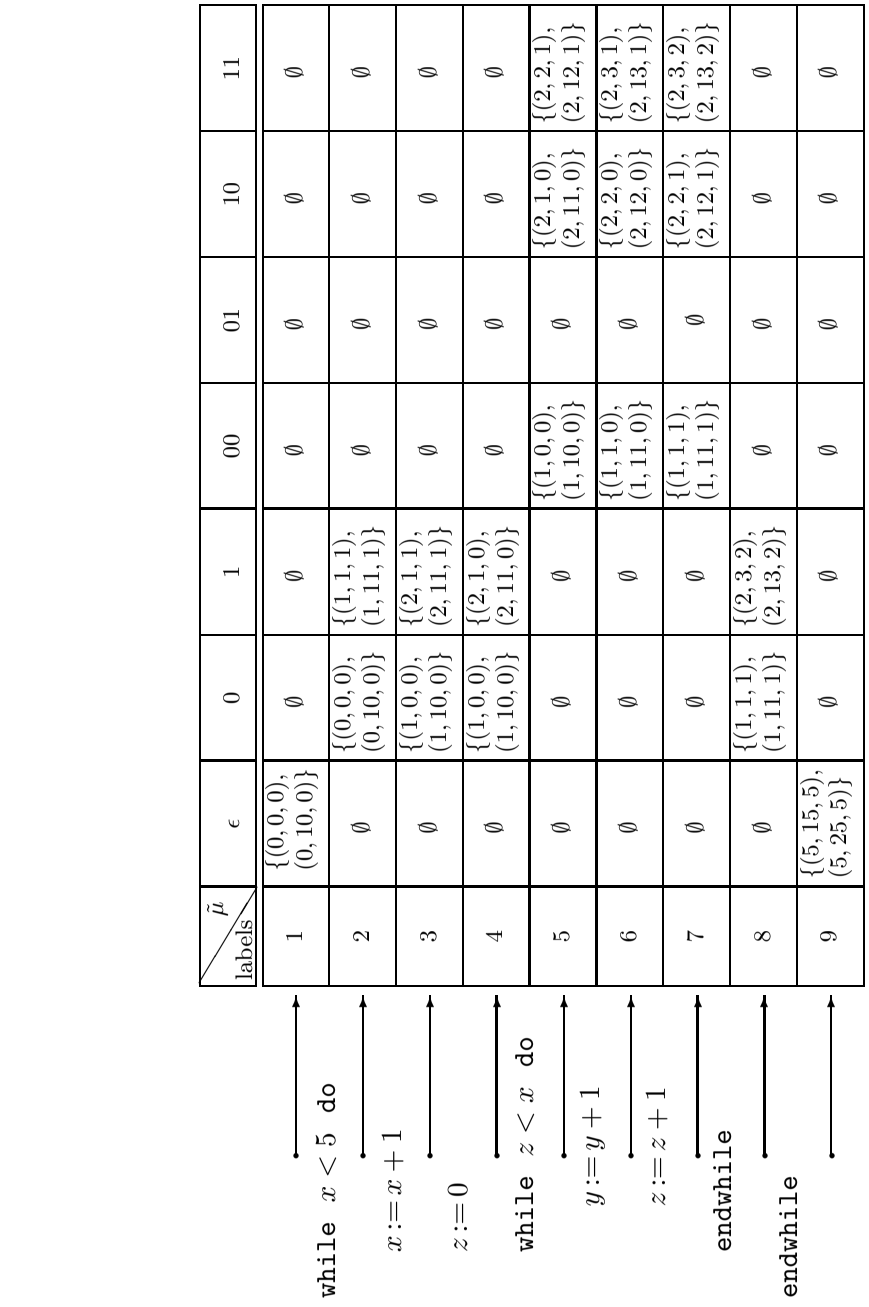
where $l_{is} = \mathit{first}(K_i),\ l_{if} = \mathit{last}(K_i),\ i \in \{1, 2\}$

$$T^p\left(\begin{array}{l}\langle l_s, \Psi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad K \\ \texttt{endwhile} \\ \langle l_f, \Psi_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \Psi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \langle l'_s, \mathit{filter}(C, \mathit{before}(\Psi_s, K|_{l'_f}))\rangle \fatsemi T^p(K) \\ \texttt{endwhile} \\ \langle l_f, \mathit{filter}(\neg C, \Psi_s \cup \mathit{collect}(K|_{l'_f}))\rangle\end{array}$$

where $l_s = \mathit{first}(K),\ l_f = \mathit{last}(K)$

$$T^p\left(K_1 \fatsemi K_2\right) = T^p\left(K_1\right) \fatsemi T^p\left(K_2\right)$$

Figure 9.1: Progressive Transfer Function



labels \ $\tilde{\mu}$	ϵ	0	1	00	01	10	11
1	$\{(0,0,0),(0,10,0)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{(0,0,0),(0,10,0)\}$	$\{(1,1,1),(1,11,1)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\{(1,0,0),(1,10,0)\}$	$\{(2,1,1),(2,11,1)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\{(1,0,0),(1,10,0)\}$	$\{(2,1,0),(2,11,0)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{(1,0,0),(1,10,0)\}$	$\emptyset$	$\{(2,1,0),(2,11,0)\}$	$\{(2,2,1),(2,12,1)\}$
6	$\emptyset$	$\emptyset$	$\emptyset$	$\{(1,1,0),(1,11,0)\}$	$\emptyset$	$\{(2,2,0),(2,12,0)\}$	$\{(2,3,1),(2,13,1)\}$
7	$\emptyset$	$\emptyset$	$\emptyset$	$\{(1,1,1),(1,11,1)\}$	$\emptyset$	$\{(2,2,1),(2,12,1)\}$	$\{(2,3,2),(2,13,2)\}$
8	$\emptyset$	$\{(1,1,1),(1,11,1)\}$	$\{(2,3,2),(2,13,2)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
9	$\{(5,15,5),(5,25,5)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 9.2: Example of Progression

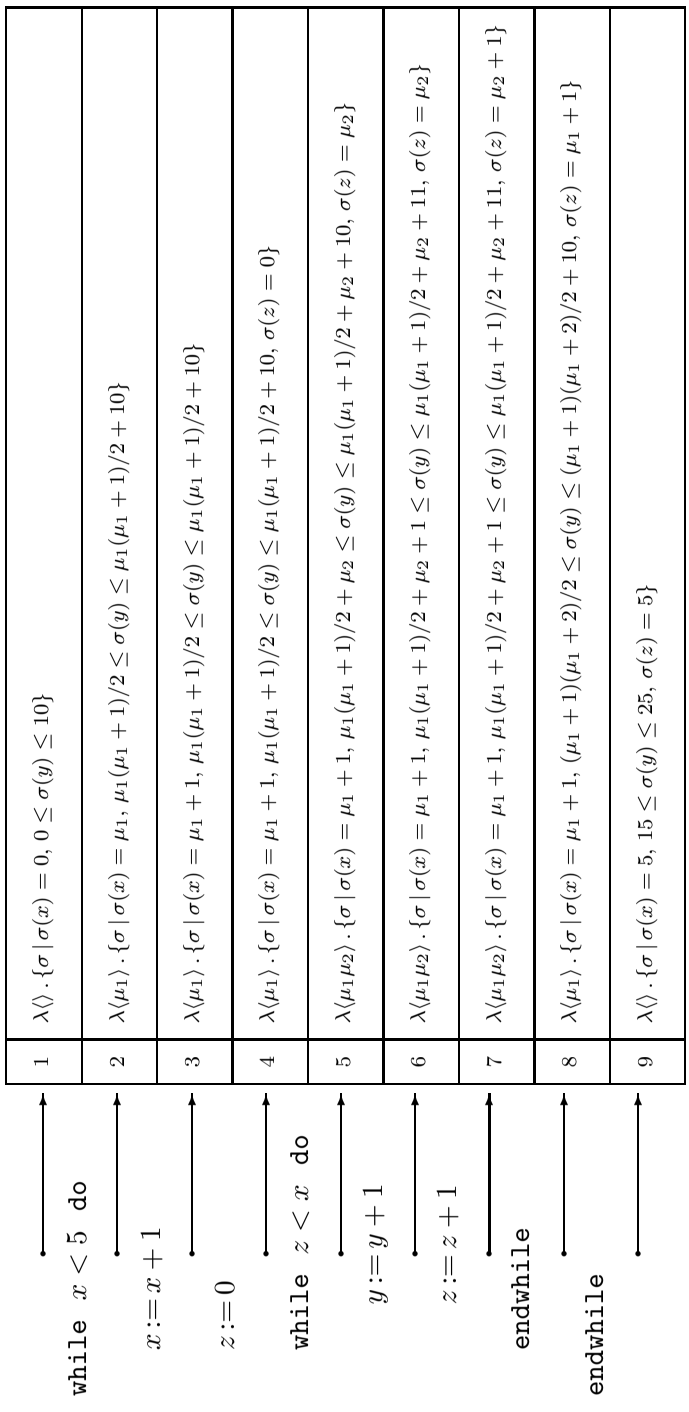


Figure 9.3: Example of Progression Using the Indexed Set Language

9.4 Proposition Let P be the sequence program $P_1 ; P_2$, and let K be a progressive configuration whose skeleton is P. Denote by K_1 and K_2 the sub-configurations of K whose skeletons are P_1 and P_2, respectively. If $T^p(K) = K$, then $T^p(K_1) = K_1$ and $T^p(K_2) = K_2$.

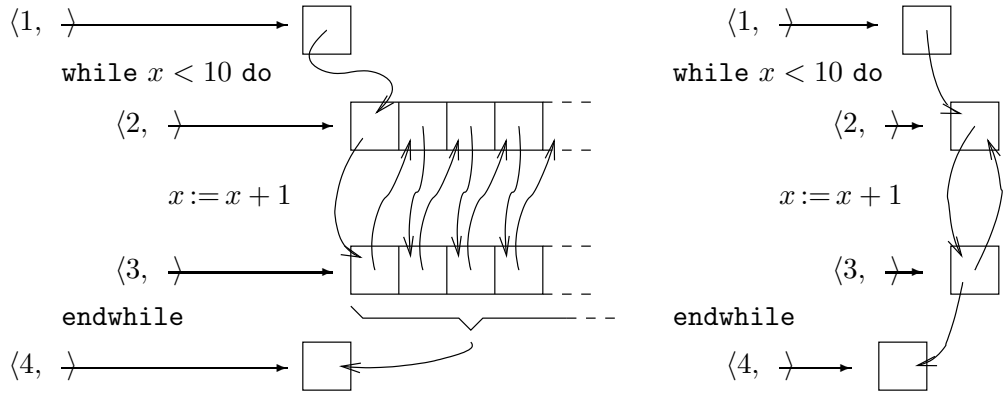
Proof: We have that $T^p(K_1 ; K_2) = K_1 ; K_2$. According to the definition of T^p, $T^p(K_1 ; K_2) = T^p(K_1) ; T^p(K_2)$. Since $|T^p(K_1)| = |K_1| = |P_1|$ and $|T^p(K_2)| = |K_2| = |P_2|$, it follows that $T^p(K_1) = K_1$ and $T^p(K_2) = K_2$. □

9.5 Proposition Let P be the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$, and let K be a progressive configuration whose skeleton is P. Denote by K_c and K_a the sub-configurations of K whose skeletons are P_c and P_a, respectively. If $T^p(K) = K$, then $T^p(K_c) = K_c$ and $T^p(K_a) = K_a$.

Proof: We have that $T^p(K) = \langle l_s, K|_{l_s} \rangle$ `if` C `then` $\langle \mathit{first}(P_c), \mathit{filter}(C, K|_{l_s}) \rangle ;$ $T^p(K_c)$ `endif` $\langle \mathit{first}(P_a), \mathit{filter}(\neg C, K|_{l_s}) \rangle ; T^p(K_a)$ `endif` $\langle l_f, K_c|_{\mathit{last}(P_c)} \cup K_a|_{\mathit{last}(P_a)} \rangle$. It follows that $K_c = \langle \mathit{first}(P_c), \mathit{filter}(C, K|_{l_s}) \rangle ; T^p(K_c)$ and $K_a = \langle \mathit{first}(P_a), \mathit{filter}(\neg C, K|_{l_s}) \rangle ; T^p(K_a)$. Since $T^p(K')|_{\mathit{first}(K')} = K'|_{\mathit{first}(K')}$, for all progressive configurations K', this entails that $T^p(K_c) = K_c$ and $T^p(K_a) = K_a$.

□

9.6 Proposition Let P be the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$, and let K be a progressive configuration whose skeleton is P. Denote by K_b the sub-configuration of K whose skeleton is P_b, respectively. If $T^p(K) = K$, then $T^p(K_b) = K_b$.



a) Refinement using T^p b) Refinement using T

Figure 9.4: Unique Fixpoint of T^p

Proof: We have that $T^p(K) = \langle l_s, K|_{l_s}\rangle$ `while` C `do` $\langle \mathit{first}(P_b), \mathit{filter}(C, K|_{l_s} \cup K_b|_{\mathit{last}(P_b)})\rangle ; T^p(K_b)$ `endwhile` $\langle l_f, \mathit{filter}(\neg C, K|_{l_s} \cup K_a|_{\mathit{last}(P_a)})\rangle$. It follows that $K_b = \langle \mathit{first}(P_b), \mathit{filter}(C, K|_{l_s} \cup K_b|_{\mathit{last}(P_b)})\rangle ; T^p(K_b)$. Since $T^p(K')|_{\mathit{first}(K')} = K'|_{\mathit{first}(K')}$, for all progressive configurations K', this entails that $T^p(K_b) = K_b$. □

9.7 Theorem Let P be a program and Σ_0 a set of start environments. There exists a unique progressive configuration K such that $|K| = P$, $K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $T^p(K) = K$.

Proof: The proof is by induction on the structure of the program P. Assume first that P is the `skip` program $\langle l_s\rangle$ `skip` $\langle l_f\rangle$. The progressive configuration w.r.t. Σ_0 is $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0\rangle$ `skip` $\langle l_f, \lambda\langle\rangle \,.\, \Sigma_0\rangle$. Obviously, $T^p(K) = K$ and K is unique for a given Σ_0. Assume now that P is the assignment $\langle l_s\rangle\, x := E\, \langle l_f\rangle$. The progressive configuration w.r.t. Σ_0 is $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0\rangle\, x := E\, \langle l_f, \mathit{assign}(x, E, \lambda\langle\rangle \,.\, \Sigma_0)\rangle$, which is unique for a given Σ_0.

Assume now that P is the sequence program $P_1 ; P_2$. Let K be a progressive configuration such that $|K| = P$, $K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $T^p(K) = K$. From the condition that $|K| = P$ it follows that K can be written as $K_1 ; K_2$, where $|K_1| = P_1$ and $|K_2| = P_2$. According to Proposition 9.4 we have that $T^p(K_1) = K_1$ and $T^p(K_2) = K_2$. According to the induction hypothesis, K_1 and K_2 are unique, and therefore so is K. Assume now that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. Let K be a progressive configuration such that $|K| = P$, $K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $T^p(K) = K$. From the condition that $|K| = P$ it follows that K can be written as $\langle l_s, \lambda\langle\rangle \,.\, \Sigma_0 \rangle$ `if` C `then` K_c `else` K_a `endif` $\langle l_f, K|_{l_f} \rangle$. Since $T^p(K) = K$, it follows from Proposition 9.5 that $T^p(K_c) = K_c$ and $T^p(K_a) = K_a$. According to the induction hypothesis, K_c and K_a are unique, and therefore so is K. Finally, assume that P is the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$. Let K be a progressive configuration such that $|K| = P$, $K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $T^p(K) = K$. From the condition that $|K| = P$ it follows that K can be written as $\langle l_s, \lambda\langle\rangle \,.\, \Sigma_0 \rangle$ `while` C `do` K_b `endwhile` $\langle l_f, K|_{l_f} \rangle$. Since $T^p(K) = K$, it follows from Proposition 9.6 that $T^p(K_b) = K_b$. According to the induction hypothesis, K_b is unique, and therefore so is K. □

This result shows that the progression K of a program P is a fixpoint of T^p, and is unique with respect to $K|_{\mathit{first}(K)}$. For this reason, whenever it is convenient, we shall call *progression of a program* P any configuration K whose skeleton is P, and which is a fixpoint of T^p, taking the liberty to disregard the set of start environments that makes K unique.

An intuitive justification this result can be provided by looking at the example in Figure 9.4, which presents a simple program that increments the variable x up to the value 10. Assume that the initial value of x at program point 1 is 0. Figure 9.4a

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma \in \mathbf{Env}\}\rangle$
  $x := 0$
$\langle 3, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 0\}\rangle$
  `while` $x < 10$ `do`
    $\langle 4, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = \mu \text{ or } \sigma(x) = 20\}\rangle$
      `if` $x < 10$ `then`
        $\langle 5, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = \mu\}\rangle$
          $x := x + 1$
        $\langle 6, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = \mu + 1\}\rangle$
      `else`
        $\langle 7, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = 20\}\rangle$
          `skip`
        $\langle 8, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = 20\}\rangle$
      `endif`
    $\langle 9, \lambda\langle\mu\rangle \,.\, \{\sigma \,|\, \sigma(x) = \mu + 1 \text{ or } \sigma(x) = 20\}\rangle$
  `endwhile`
$\langle 10, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 10 \text{ or } \sigma(x) = 20\}\rangle$

Figure 9.5: A Post-Fixpoint K of T^p

provides a graphic representation of the refinement process using the T^p operator, while Figure 9.4b represents the same process using the T operator. As argued in the previous chapter, the T operator has, in general, multiple fixpoints. In the case of this program, the greatest fixpoint is a configuration with the set of environments $\{\sigma \,|\, \sigma(x) < 10\}$ attached to program point 3. The process of refinement using the T operator follows the flow graph of the program, which has a loop between program points 1 and 2. for this reason, the set $\{\sigma \,|\, \sigma(x) \leq 0\}$, which is not part of the collecting semantics, cannot be eliminated in the process of refinement. However, refinement using the T^p operator applied to a progressive configuration has the advantage of using indexed sets and effectively breaking the loop that appears in

the flow graph of the program, as shown in Figure 9.4a. Indeed, in the refinement process, the information in the first slice of the indexed set at program point 2 is propagated to the first slice of program point 3. However, that information is further propagated into the second slice of program point 2. Every propagation from program point 3 to program point 2 is shifted by one slice by the *before* operator. Thus, at fixpoint, the information in every slice depends solely on the information at the first program point, which explains why the fixpoint of T^p w.r.t. a given set of start environments is unique. It may be expected that repeated applications of the T^p operator to a progressive configuration shall produce a sequence of progressive configurations that converges towards the least fixpoint. However, as we show in the next section, $T^p \downarrow \omega$ is not always a fixpoint of T^p.

9.3 Refinement

In the previous section we have shown that the progression of a program P w.r.t. a set of start environments Σ_0 is the unique fixpoint of the T^p operator over the lattice $(\{K \,|\, |K| = P \text{ and } K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0\}, \preccurlyeq)$. One may think that the absence of other fixpoints allows the T^p operator to be used in computing arbitrarily precise approximations of progressions of a program P. However, repeated applications of the T^p (or any monotone operator) to one of its postfixpoints K will produce approximations that cannot be more precise than $(T^p)^\omega(K)$. It turns out that $(T^p)^\omega(K)$ is not necessarily a fixpoint of T^p, and therefore the amount of refinement one can do by repeated applications of the T^p operator is limited and cannot produce arbitrarily precise approximations.

The following proposition shows, however, that repeated applications of T^p monotonically increase the precision of an approximation. The proof provides an

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma \in \mathbf{Env}\}\rangle$
  $x := 0$
$\langle 3, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma(x) = 0\}\rangle$
  `while` $x < 10$ `do`
    $\langle 4, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu\}\rangle$
      `if` $x < 10$ `then`
        $\langle 5, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu\}\rangle$
          $x := x + 1$
        $\langle 6, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu + 1\}\rangle$
      `else`
        $\langle 7, \lambda\langle\mu\rangle \,.\, \emptyset\rangle$
          `skip`
        $\langle 8, \lambda\langle\mu\rangle \,.\, \emptyset\rangle$
      `endif`
    $\langle 9, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu + 1\}\rangle$
  `endwhile`
$\langle 10, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma(x) = 10 \text{ or } \sigma(x) = 20\}\rangle$

Figure 9.6: $(T^p)^\omega(K)$ for the configuration K in Figure 9.5

insight of how information is propagated inside a progressive configuration by the T^p operator.

9.8 Proposition Given a program P and a set of start environments Σ_0, denote by K^{PS} the progression of P w.r.t. Σ_0, and let K be a progressive configuration such that $|K| = P$, $K|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $K^{PS} \preccurlyeq K$. Then, for all $k \geq 0$, there exists a set $A = \{(l_0, \tilde{\mu}_0), (l_1, \tilde{\mu}_1), \ldots, (l_k, \tilde{\mu}_k)\}$ of pairs of labels and indices such that $K^{CS}|_{l_i}(\tilde{\mu}_i) = (T^p)^k(K)|_{l_i}(\tilde{\mu}_i)$, $0 \leq i \leq k$.

Proof: The proof is by induction on k. For $k = 0$, we have that A is the set $\{(\mathit{first}(P), 0)\}$. Indeed, $K^{CS}|_{\mathit{first}(P)}(0) = K|_{\mathit{first}(P)}(0)$, Assume now that the proposition holds for some natural number k. Denote by A the set $\{(l_0, \tilde{\mu}_0), \ldots, (l_k, \tilde{\mu}_k)\}$

such that $K^{CS}|_{l_i}(\tilde{\mu}_i) = (T^p)^k(K)|_{l_i}(\tilde{\mu}_i)$, $0 \leq i \leq k$. Since A is a finite set, at least one of the following situations will happen.

a) $\langle l_s, \Psi_s \rangle$ `skip` $\langle l_f, \Psi_f \rangle$ is a sub-configuration of K and there exists an index $\tilde{\mu}$ such that $(l_s, \tilde{\mu}) \in A$ and $(l_f, \tilde{\mu}) \notin A$.

b) $\langle l_s, \Psi_s \rangle\, x := E\, \langle l_f, \Psi_f \rangle$ is a sub-configuration of K and there exists an index $\tilde{\mu}$ such that $(l_s, \tilde{\mu}) \in A$ and $(l_f, \tilde{\mu}) \notin A$.

c) There exist configurations K_c and K_a such that $\langle l_s, \Psi_s \rangle$ `if` C `then` K_c `else` K_a `endif` $\langle l_f, \Psi_f \rangle$ is a sub-configuration of K, and there exist an index $\tilde{\mu} \in \mathbf{Idx}$ such that either $(l_s, \tilde{\mu}) \in A$ and $(\mathit{first}(K_c), \tilde{\mu}) \notin A$, or $(l_s, \tilde{\mu}) \in A$ and $(\mathit{first}(K_a), \tilde{\mu}) \notin A$, or $(\mathit{last}(K_c), \tilde{\mu}) \in A$ and $(l_f, \tilde{\mu}) \notin A$ or $(\mathit{last}(K_a), \tilde{\mu}) \in A$ and $(l_f, \tilde{\mu}) \notin A$.

d) There exists a configuration K_b such that $\langle l_s, \Psi_s \rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_f \rangle$ is a sub-configuration of K, and there exists an index $\tilde{\mu} \in \mathbf{Idx}$ such that either $(l_s, \tilde{\mu}) \in A$ and $(\mathit{first}(K_b), \tilde{\mu}0) \notin A$, or $(l_s, \tilde{\mu}) \in A$ and $(l_f, \tilde{\mu}) \notin A$, or $(\mathit{last}(K_b), \tilde{\mu}) \in A$ and $(\mathit{first}(K_b), \mathit{succ}(\tilde{\mu})) \notin A$, where $\mathit{succ}(\tilde{\mu})$ is the successor of $\tilde{\mu}$.

Applying the definition of T^p, it is easy to verify that in each of the 4 cases given above, $T^p((T^p)^k(K))$ adds one more element to A. Therefore, for $(T^p)^{k+1}(K)$ there exist a set of at least $k+2$ elements $\{(l_i, \tilde{\mu}_i) \mid 0 \leq i \leq k+1\}$ such that $K^{CS}|_{l_i}(\tilde{\mu}_i) = (T^p)^k(K)|_{l_i}(\tilde{\mu}_i)$. □

To give more insight into refining configurations using the T^p operator, consider the configuration K given in Figure 9.5. This configuration is a postfixpoint of T^p. We note that the integer 20, which appears as a possible value of x at program points 4, 7, 8, 9, and 10 does not really occur during the execution of the program.

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma \in \mathbf{Env}\}\rangle$
  $x := 0$
$\langle 3, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma(x) = 0\}\rangle$
  `while` $x < 10$ `do`
    $\langle 4, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu\}\rangle$
      `if` $x < 10$ `then`
        $\langle 5, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu\}\rangle$
          $x := x + 1$
        $\langle 6, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu + 1\}\rangle$
      `else`
        $\langle 7, \lambda\langle\mu\rangle \,.\, \emptyset\rangle$
          `skip`
        $\langle 8, \lambda\langle\mu\rangle \,.\, \emptyset\rangle$
      `endif`
    $\langle 9, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \mu + 1\}\rangle$
  `endwhile`
$\langle 10, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma(x) = 10\}\rangle$

Figure 9.7: Progression of Program in Figure 9.5

It might be expected that repeated applications of T^p to this configuration would remove 20 as a possible value of x from all program points and all indexed set slices. However, this is not true. Figure 9.6 shows the configuration $(T^p)^\omega(K)$. We notice the fact that, while the value 20 has disappeared from program points 4, 7, 8, and 9, it is still present at program point 10. Figure 9.7 represents the progression of the program that is the skeleton of K, w.r.t. the set of start environments **Env**. We also note the fact that the progression of $|K|$ is in fact $T^p((T^p)^\omega(K))$.

Chapter 10

The Semantics Hierarchy

In this chapter we show formally that the progressive semantics' level of abstraction is in between that of the trace semantics and that of the collecting semantics. This result is achieved by proving that the progressive semantics is an abstract interpretation of the trace semantics, while the collecting semantics is an abstract interpretation of the progressive semantics. The idea of hierarchizing a set of semantics using abstract interpretation has been introduced in [Cou02]. We proceed by showing that the three domains we used for the three semantics presented so far are Galois connected. We devise two abstraction mappings $\vec{\alpha}$ and $\overline{\alpha}$ and two concretization mappings $\vec{\gamma}$ and $\overline{\gamma}$ such that the following Galois insertions exist:

$$2^{\mathbf{Singletons}(P)} \underset{\vec{\alpha}}{\overset{\vec{\gamma}}{\leftrightarrows}} \mathbf{Progressive}(P) \underset{\overline{\alpha}}{\overset{\overline{\gamma}}{\leftrightarrows}} \mathbf{Collecting}(P)$$

To complete the picture we also show that $T^p = \vec{\alpha} \circ \vec{T} \circ \vec{\gamma}$, and $T = \overline{\alpha} \circ T^p \circ \overline{\gamma}$. This entails that, given a program P and a set of start environments Σ_0, and denoting by $\vec{\Gamma}$, K, and $\overline{K}$ the trace, progressive and collecting semantics of P w.r.t. Σ_0, we have that $K = \vec{\alpha}(\vec{\Gamma})$ and $\overline{K} = \overline{\alpha}(K)$.

10.1 Progressive Semantics is Abstract Interpretation of Trace Semantics

Let P be a labeled program. We define the abstraction mapping $\vec{\alpha} : \bigcup_P 2^{\mathbf{Singletons}(P)} \mapsto \bigcup_P \mathbf{Progressive}(P)$ as follows. Given a set of singleton configurations Γ, $\vec{\alpha}(\Gamma)$ is a progressive configuration K such that $K|_l(\tilde{\mu}) = \bigcup_{\vec{K}\in\Gamma} \vec{K}|_l(\tilde{\mu})$, for all $l \in \mathit{labels}(P)$ and $\tilde{\mu} \in \mathbf{Idx}$. It's worth noting that given a family of sets of singleton configurations $\tilde{\Gamma}$, we have that $\vec{\alpha}(\bigcup_{\vec{\Gamma}\in\tilde{\Gamma}} \vec{\Gamma}) = \bigcup_{\vec{\Gamma}\in\tilde{\Gamma}} \vec{\alpha}(\vec{\Gamma})$, and that $\vec{\alpha}(\{\vec{K}\}) = \vec{K}$, for all singleton configurations $\vec{K}$. We also define the concretization mapping $\vec{\gamma} : \bigcup_P \mathbf{Progressive}(P) \mapsto \bigcup_P 2^{\mathbf{Singletons}(P)}$ by $\vec{\gamma}(K) = \{\vec{K} \mid \vec{K} \in \mathbf{Singletons}(|K|)$ and $\vec{K}|_l(\tilde{\mu}) \subseteq K|_l(\tilde{\mu}), l \in \mathit{labels}(|K|), \tilde{\mu} \in \mathbf{Idx}\}$. It is easy to verify that, for a singleton configuration $\vec{K}$, we have that $\vec{\gamma}(\vec{K}) = \{\vec{K}\}$.

10.1 Proposition Given a program P, and a set of configurations Γ whose skeleton is P, we have that $\bigcup_{K\in\Gamma} \vec{\gamma}(K) \subseteq \vec{\gamma}(\bigcup_{K\in\Gamma} K)$.

Proof: Let $\vec{K}$ be a member of $\bigcup_{K\in\Gamma} \vec{\gamma}(K)$. Then, there exists $K \in \Gamma$ such that $\vec{K} \in \vec{\gamma}(K)$. This actually means that for all labels $l \in \mathit{labels}(P)$ and indexes $\tilde{\mu} \in \mathbf{Idx}$ we have $\vec{K}|_l(\tilde{\mu}) \subseteq K|_l(\tilde{\mu})$, which in turn entails that $\vec{K}|_l(\tilde{\mu}) \subseteq (\bigcup_{K\in\Gamma} K)|_l(\tilde{\mu})$. According to the definition of $\vec{\gamma}$, it follows that $\vec{K} \in \vec{\gamma}(\bigcup_{K\in\Gamma} K)$. □

10.2 Proposition Given a program P, we have the Galois insertion:

$$2^{\mathbf{Singletons}(P)} \underset{\vec{\alpha}}{\overset{\vec{\gamma}}{\leftrightarrows}} \mathbf{Progressive}(P).$$

Proof: The mappings $\vec{\alpha}$ and $\vec{\gamma}$ are clearly monotonic. First we prove that $\vec{\gamma} \circ \vec{\alpha} \supseteq \lambda x\,.\,x$. This amounts to proving that $\vec{\Gamma} \subseteq \vec{\gamma}(\vec{\alpha}(\vec{\Gamma}))$ for all sets of singleton configura-

tions $\vec{\Gamma}$. Indeed, using Proposition 10.1 we have that $\vec{\gamma}(\vec{\alpha}(\vec{\Gamma})) = \vec{\gamma}(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}\})) = \vec{\gamma}(\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{\alpha}(\{\vec{K}\})) = \vec{\gamma}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}\}) \supseteq \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{\gamma}(\{\vec{K}\}) = \bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}\} = \Gamma$.

Next we prove that $\vec{\alpha}\circ\vec{\gamma} = \lambda y\,.\,y$. This amounts to proving that for all progressive configurations $K \in \mathbf{Progressive}(P)$, labels $l \in \mathit{labels}(P)$, and indices $\tilde{\mu} \in \mathbf{Idx}$, we have $\vec{\alpha}(\vec{\gamma}(K))|_l(\tilde{\mu}) = K|_l(\tilde{\mu})$. Indeed, $\vec{\alpha}(\vec{\gamma}(K))|_l(\tilde{\mu}) = \bigcup_{\sigma\in K|_l(\tilde{\mu})}\{\sigma\} = K|_l(\tilde{\mu})$. □

10.3 Remark The abstraction mapping $\vec{\alpha}$ distributes over the syntactic structure of a program. The following properties follow immediately from the definition of $\vec{\alpha}$:

a) $\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\ \texttt{skip}\ \langle l_f, \vec{K}|_{l_f}\rangle\}) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\})\ \texttt{skip}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_f}\rangle\})$.

b) $\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\ x := E\ \langle l_f, \vec{K}|_{l_f}\rangle\}) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\})\ x := E\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_f}\rangle\})$.

c) $\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst} \fatsemi \vec{K}^{snd}\}) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\}) \fatsemi \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\})$.

d) $\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\ \texttt{if}\ C\ \texttt{then}\ \vec{K}^c\ \texttt{else}\ \vec{K}^a\ \texttt{endif}\ \langle l_f, \vec{K}|_{l_f}\rangle\}) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\})\ \texttt{if}\ C\ \texttt{then}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\})\ \texttt{else}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\})\ \texttt{endif}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_f, \vec{K}|_{l_f}\rangle\})$.

e) $\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\ \texttt{while}\ C\ \texttt{do}\ \vec{K}^b\ \texttt{endwhile}\ \langle l_f, \vec{K}|_{l_f}\rangle\}) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\})\ \texttt{while}\ C\ \texttt{do}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^b\})\ \texttt{endwhile}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_f, \vec{K}|_{l_f}\rangle\})$.

□

10.4 Remark The abstraction mapping $\vec{\alpha}$ distributes over the indexed set operators defined in Figure 7.3. Let $\tilde{\Psi}$ be a family of indexed sets. Then, the following properties follow immediately from the definition of $\vec{\alpha}$.

a) $\vec{\alpha}(\{\langle l, \mathit{assign}(x, E, \Psi)\rangle \mid \Psi \in \tilde{\Psi}\}) = \langle l, \mathit{assign}(x, E, \bigcup_{\Psi\in\tilde{\Psi}}\Psi)\rangle$.

b) $\vec{\alpha}(\{\langle l, \mathit{filter}(C, \Psi)\rangle \mid \Psi \in \tilde{\Psi}\}) = \langle l, \mathit{filter}(C, \bigcup_{\Psi\in\tilde{\Psi}}\Psi)\rangle$.

c) $\vec{\alpha}(\{\langle l, \mathit{before}(\Psi, \Psi')\rangle \,|\, (\Psi, \Psi') \in \tilde{\Upsilon}\}) = \langle l, \mathit{before}(\bigcup_{(\Psi,\Psi')\in\tilde{\Upsilon}} \Psi, \bigcup_{(\Psi,\Psi')\in\tilde{\Upsilon}} \Psi')\rangle$

d) $\vec{\alpha}(\{\langle l, \mathit{collect}(\Psi)\rangle \,|\, \Psi \in \tilde{\Psi}\}) = \langle l, \mathit{collect}(\bigcup_{\Psi\in\tilde{\Psi}} \Psi)\rangle$

□

The following definition shows that T^p has a sound definition w.r.t. the trace progress operator $\vec{T}$.

10.5 Proposition We have that $T^p \circ \vec{\alpha} = \vec{\alpha} \circ \vec{T}$.

Proof: Relegated to Appendix A, on page 300.

We are now finally able to provide a result announced in Section 9.2. The following lemma states that the progression of a program P w.r.t. a set of start environments Σ_0 is the least fixpoint of the T^p operator over the lattice $(\{K \,|\, |K| = P \text{ and } K|_{\mathit{first}(P)} = \lambda\langle\rangle\,.\,\Sigma_0\}, \preccurlyeq)$.

10.6 Lemma Consider a program P and a set of start environments Σ_0, and denote by K the progression of P w.r.t. Σ_0. Then, $K = \mathit{lfp}(T^{p\prime} \cup K_0)$, where $T^{p\prime}$ is the restriction of T^p to the set $\{K \,|\, |K| = P\}$, and K_0 is the configuration defined by $K_0|_{\mathit{first}(P)} = \lambda\langle\rangle\,.\,\Sigma_0$ and $K_0|_l = \lambda\langle\rangle\,.\,\emptyset$ for all $l \in \mathit{labels}(P)\backslash\{l\}$.

Proof: Let $\vec{\Gamma}$ be the configuration-based trace semantics of P w.r.t. Σ_0. From the definition of $\vec{\alpha}$, we have that $K = \vec{\alpha}(\vec{\Gamma})$. It is also immediate to prove that if $T^p \circ \vec{\alpha} = \vec{\alpha} \circ \vec{T}$, then $(T^p \cup K_0) \circ \vec{\alpha} = \vec{\alpha} \circ (\vec{T} \cup \vec{\Gamma}_0)$, where $\vec{\Gamma}_0$ is the set of singleton configurations $\{\vec{K} \,|\, |\vec{K}| = P,\ \vec{K}_0|_{\mathit{first}(P)} = \lambda\langle\rangle\,.\,\{\sigma\},\ \sigma \in \Sigma_0, \vec{K}_0|_l = \lambda\langle\rangle\,.\,\emptyset \text{ for all } l \in \mathit{labels}(P)\backslash\{l\}\}$. Remark 9.2 states that K is the fixpoint of the restriction $T^{p\prime}$ of T^p to the set $\{\vec{K} \,|\, |\vec{K}| = P\}$. From Propositions 10.5 and 2.11 it follows immediately that $K = \mathit{lfp}(T^{p\prime} \cup K_0)$. □

We now complete the picture by outlining the link between the trace and the progressive semantics.

10.7 Lemma Consider a program P and a set of start environments Σ_0, and denote by $\vec{\Gamma}$ and K the trace semantics and the progression of P w.r.t. Σ_0, respectively. Then, $K = \vec{\alpha}(\vec{\Gamma})$.

Proof: From Propositions 5.7 and 7.13 we have that the trace semantics $\vec{\Gamma}$ is a fixpoint of the $\vec{T}$ operator, and from Lemma 10.6 we have that K is a fixpoint of the T^p operator. Theorem 10.5 states that $T^p \circ \vec{\alpha} = \vec{\alpha} \circ \vec{T}$. This entails that $(\vec{\alpha} \circ \vec{T})(\vec{\Gamma}) = \vec{\alpha}\vec{\Gamma} = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. It follows that $\vec{\alpha}(\vec{\Gamma})$ is a fixpoint of T^p. According to Theorem 9.7, the fixpoint of T^p is unique for a given program P and a given set of start environments Σ_0, and therefore it follows that $\vec{\alpha}(\vec{\Gamma}) = K$. □

10.2 Collecting Semantics is Abstract Interpretation of Progressive Semantics

We now define the abstraction mapping $\overline{\alpha}$: **Progressive**$(P) \mapsto$ **Collecting**(P) as follows. Given a progressive configuration K, $\overline{\alpha}(K)$ is a collective configuration $\overline{K}$ such that $\overline{K}|_l(\tilde{\mu}) = \bigcup_{\tilde{\mu}} K|_l(\tilde{\mu})$. We also define the concretization mapping $\overline{\gamma}$: **Collecting**$(P) \mapsto$ **Progressive**(P) as follows. Given a collective configuration $\overline{K}$, $\overline{\gamma}(\overline{K})$ is a progressive configuration K such that $K|_l(\tilde{\mu}) = \overline{K}|_l$ for all $l \in \mathit{labels}(P)$ and $\tilde{\mu} \in$ **Idx**.

10.8 Proposition Given a program P, we have the following Galois insertion:

$$\textbf{Progressive}(P) \underset{\overline{\alpha}}{\overset{\overline{\gamma}}{\leftrightarrows}} \textbf{Collecting}(P).$$

Proof: The mappings $\overline{\alpha}$ and $\overline{\gamma}$ are clearly monotonic. First we prove that $\overline{\gamma}_P \circ \overline{\alpha} \supseteq \lambda x\,.\,x$. This amounts to proving that $\overline{\gamma}_P(\overline{\alpha}(K))|_l \supseteq K|_l$, for all $K \in \mathbf{Progressive}(P)$, and $l \in \mathit{labels}(P)$. According to the definitions of $\overline{\alpha}$ and $\overline{\gamma}$, we have that $\overline{\gamma}_P(\overline{\alpha}(K))|_l = \bigcup_{\tilde{\mu}} K|_L(\tilde{\mu})$, which satisfies the above property. Next we prove that $\overline{\alpha} \circ \overline{\gamma}_P = \lambda y\,.\,y$. This amounts to proving that for all collective configurations $\overline{K}$, and all labels $l \in \mathit{labels}(l)$ we have $\overline{\alpha}(\overline{\gamma}(\overline{K}))|_l = \overline{K}|_l$, which follows immediately from the definitions of $\overline{\alpha}$ and $\overline{\gamma}$. □

10.9 Remark The abstraction mapping $\overline{\alpha}$ distributes over the syntactic structure of a program. The following properties follow immediately from the definition of $\overline{\alpha}$:

a) $\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle \texttt{ skip } \langle l_f, \vec{K}|_{l_f}\rangle = \overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle) \texttt{ skip } \overline{\alpha}(\langle l_s, \vec{K}|_{l_f}\rangle)$.

b) $\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle\ x := E\ \langle l_f, \vec{K}|_{l_f}\rangle) = \overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle)\ x := E\ \overline{\alpha}(\langle l_s, \vec{K}|_{l_f}\rangle)$.

c) $\overline{\alpha}(\vec{K}^{fst} \fatsemi \vec{K}^{snd}) = \overline{\alpha}(\vec{K}^{fst}) \fatsemi \overline{\alpha}(\vec{K}^{snd})$.

d) $\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle \texttt{ if } C \texttt{ then } \vec{K}^c \texttt{ else } \vec{K}^a \texttt{ endif } \langle l_f, \vec{K}|_{l_f}\rangle) =$
$\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle) \texttt{ if } C \texttt{ then } \overline{\alpha}(\{\vec{K}^c) \texttt{ else } \overline{\alpha}(\vec{K}^a) \texttt{ endif } \overline{\alpha}(\langle l_f, \vec{K}|_{l_f}\rangle)$.

e) $\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle \texttt{ while } C \texttt{ do } \vec{K}^b \texttt{ endwhile } \langle l_f, \vec{K}|_{l_f}\rangle) =$
$\overline{\alpha}(\langle l_s, \vec{K}|_{l_s}\rangle) \texttt{ while } C \texttt{ do } \overline{\alpha}(\vec{K}^b) \texttt{ endwhile } \overline{\alpha}(\langle l_f, \vec{K}|_{l_f}\rangle)$.

□

10.10 Remark The abstraction mapping $\overline{\alpha}$ distributes over the indexed set operators defined in Figure 7.3. Let $\tilde{\Psi}$ be a family of indexed sets. Then, the following properties follow immediately from the definition of $\overline{\alpha}$.

a) $\overline{\alpha}(\langle l, \mathit{assign}(x, E, \Psi)\rangle) = \langle l, \{\sigma[x \mapsto E] \,|\, \sigma \in \bigcup_{\tilde{\mu}} \Psi(\tilde{\mu})\}\rangle$.

b) $\overline{\alpha}(\langle l, \mathit{filter}(C, \Psi)\rangle) = \langle l, \{\sigma \,|\, \sigma \models C\} \cap \bigcup_{\tilde{\mu}} \Psi\rangle$.

c) $\overline{\alpha}(\langle l, \mathit{before}(\Psi, \Psi')\rangle) = \langle l, \bigcup_{\tilde{\mu}} \Psi(\tilde{\mu}) \cup \bigcup_{\tilde{\mu}} \Psi'(\tilde{\mu})\rangle$.

d) $\overline{\alpha}(\langle l, \mathit{collect}(\Psi)\rangle) = \langle l, \bigcup_{\tilde{\mu}} \Psi(\tilde{\mu})\rangle$.

□

10.11 Proposition Given a program P, we have that $T \circ \overline{\alpha} = \overline{\alpha} \circ T^p$.

Proof: Relegated to Appendix A, on page 306.

We now complete the picture by outlining the link between the progressive and collecting semantics.

10.12 Lemma Consider a program P and a set of start environments Σ_0, and denote by K and $\overline{K}$ the progression and the collective semantics of P w.r.t. Σ_0, respectively. Then, $\overline{K} = \overline{\alpha}(K)$.

Proof: Denote by K_0 the progressive configuration defined by $K_0|_{\mathit{first}(P)} = \lambda\langle\rangle \,.\, \Sigma_0$ and $K_0|_l = \lambda\langle\rangle \,.\, \emptyset$ for all $l \in \mathit{labels}(P)\backslash\{l\}\}$, and by $\overline{K}_0$ the collective configuration defined by $\overline{K}_0|_{\mathit{first}(P)} = \Sigma_0$ and $\overline{K}_0|_l = \emptyset$ for all $l \in \mathit{labels}(P)\backslash\{l\}\}$. It is immediate to prove that $(T \cup \overline{K}_0) \circ \overline{\alpha} = \overline{\alpha} \circ (T^p(K) \cup K_0)$, if $T \circ \overline{\alpha} = \overline{\alpha} \circ T^p$. Using Propositions 8.3 and 10.11 and Remark 9.2 the result follows immediately. □

Throughout this chapter we have outlined the relationship between the trace, progressive and collecting semantics, by defining appropriate abstraction and concretization operators and showing that the progressive semantics can be seen as an abstract interpretation of the trace semantics, while the collecting semantics can

be seen as an abstract interpretation of the progressive semantics. However, this formal argument does not emphasize the fact that the progressive semantics has the *right* level of abstraction, making it possible to reason about liveness and safety properties in a compositional, program-point-based manner. We shall outline these features in the next part of this thesis, where we provide a treatment of progressive program reasoning methods.

Part III

Progressive Program Reasoning

Chapter 11

Family Configurations

The collecting semantics is a useful abstraction of a program's behavior, as it expresses a property of all states occurring during the program's execution. Given a program P, if one were able to compute its collecting semantics CS w.r.t. some set of start states Σ_0, then one could expose properties of the program; for instance, it could be detected whether the program at hand satisfies a safety property, or even if the program terminates, by inspecting whether the projection of the collecting semantics CS on the last program point is the empty set. However, calculating the collecting semantics of a program may require a great deal of effort and a high level of expertise. In practice we usually resort to producing an approximation of the collecting semantics, i.e. an object whose interpretation is in a well-defined relationship with the collecting semantics CS. Classic program reasoning methods typically produce as an approximation a superset RS of the collecting semantics CS. The approximation RS is interpreted in the following way: any subset of RS, including the empty set, could be the collecting semantics of the program P. Such a relationship is strong enough for proving *safety properties*, since any property of the states in the set RS is also a property of the states in CS. However, proving

progress properties may require the ability to explicitly infer that a projection of CS on a particular program point is not empty, or that some state in CS occurs before another.

The superset relationship between RS and CS is too weak to capture such properties; on one hand, the collecting semantics abstracts away the sequence in which the states occur, and on the other hand, the superset (*conservative*) approximation allows too wide a range of possible values for the collecting semantics.

We can overcome the shortcomings mentioned above by using the following two steps. First, we shall use the progressive semantics as the basis for our reasoning. The progressive semantics captures an abstraction of the sequencing of the states that is strong enough to capture progress properties. Second, given a set of environments Σ, we shall approximate it by a *family* of sets of environments Φ, such that $\Sigma \in \Phi$. Approximating by this method allows more flexibility in specifying the range of possible values for the environment Σ. In particular, if the family Φ does not contain the empty set, it means that Σ cannot be empty, and therefore, the program point to which the set of environments Σ is attached is *live*.

In this chapter, we introduce *family configurations*, which are essentially configurations whose annotations are families of sets of environments, and then show that such configurations are convenient means of approximating the progressive semantics, leading to more expressive power than classic program reasoning methods. In particular, safety, liveness and progress properties can be expressed and derived within an unified framework. Moreover, richer properties can be expressed. For example, given a setting where a computing system has a limited amount of cache, it may be interesting to verify that a certain variable will only have a limited number of distinct values, irrespective of how long the program runs. Such a variable may be used as the index of an array, and the limited number of values of the

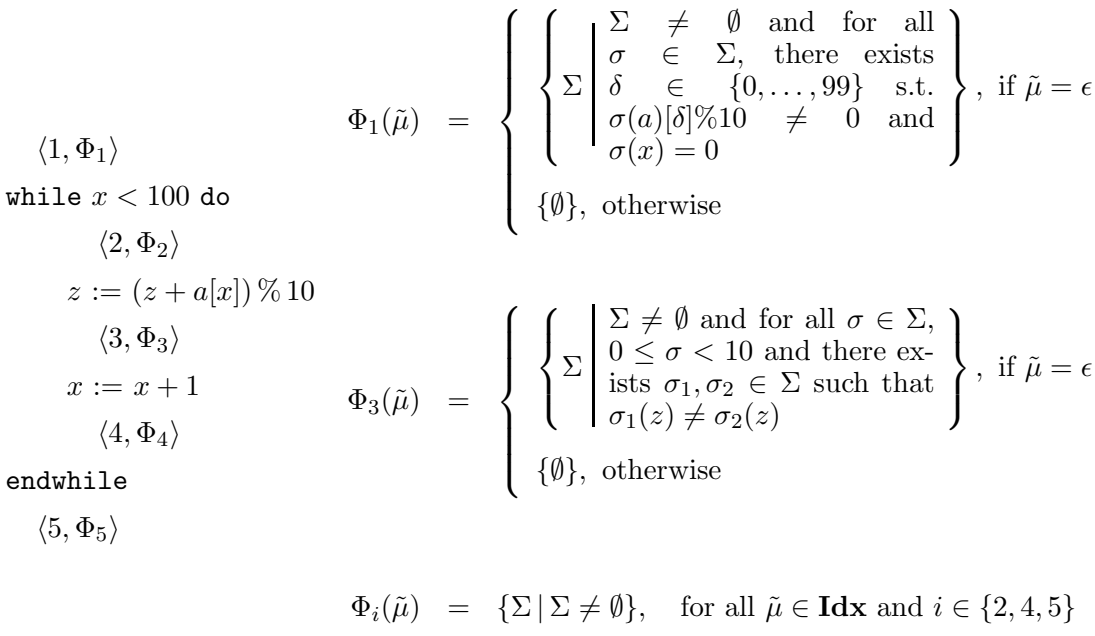


Figure 11.1: Example: number of distinct values for a variable

index guarantees that all the array elements that will ever be accessed fit entirely in the cache. Such a situation is shown in the program given in Figure 11.1. This program performs a modulo 10 summation of the elements between 0 and 99 of the array a, and the sum is accumulated in the variable z. It follows that at program point $\langle 3 \rangle$ variable z will *definitely* have *at least* two distinct values (and at most ten), provided that at program point $\langle 1 \rangle$ the array a contains at least one value that is not a multiple of 10, and that variable x has the value 0. These properties at program points $\langle 1 \rangle$ and $\langle 3 \rangle$ can be expressed by the families Φ_1 and Φ_3 in Figure 11.1. We note that the property expressed by Φ_3 could not be expressed in classic, collecting-semantics-based program reasoning methods.

11.1 Families and Family Configurations

This section defines families and family configurations as means of approximating the progressive semantics. When we speak of approximating the progressive semantics, we have to consider two somewhat independent directions: approximating the sequencing of the states that occur during the execution of the program, and approximating slices of indexed sets. The first direction is already embedded in the definition of the progressive semantics.

As defined in Chapter 9, the progression of a program P is parameterized by a set of start environments Σ_0. The indexed set attached to the first program point of P is $\lambda\langle\rangle\,.\,\Sigma_0$. In a more general setting, however, keeping in mind that the program P may in fact be a fragment of a larger program, it may be useful to allow the progression to be parameterized by any indexed set. In other words, we may want to assume that the program fragment P may be executed multiple times as part of a loop that resides in a larger program P'. Or, we may simply wish to distinguish between several categories of start environments and provide a more refined progression of the program. Such situation is illustrated by the example given in Figure 11.2. This example shows three progressions of the same program fragment. The program fragment simply increments a variable by 2 inside a loop. Figure 11.2a shows the least refined progression K_1. Due to the fact that we did not provide a mechanism to distinguish between the various variables of x, it is not possible to infer any information about, for instance, the parity of x, or the sequence in which the values of x occur. The first attempt at providing more refined information is done in Figure 11.2b, where the first program point of configuration K_2 has an indexed set annotation which distinguishes between odd and even values of x. It is easy to check that $\mathit{collect}(K_1|_{\mathit{first}(P)}) = K_2|_{\mathit{first}(P)}$. In

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \mid 0 \leq \sigma(x) < \sigma(n)\}\rangle$

  `while` $x < n$ `do`

    $\langle 2, \lambda\langle\mu\rangle \,.\, \{\sigma \mid 2\mu \leq \sigma(x) < \sigma(n)\}\rangle$

      $x := x + 2$

    $\langle 3, \lambda\langle\mu\rangle \,.\, \{\sigma \mid 2\mu \leq \sigma(x) - 2 < \sigma(n)\}\rangle$

  `endwhile`

$\langle 4, \lambda\langle\rangle \,.\, \{\sigma \mid \sigma(n) \leq \sigma(x) < \sigma(n) + 2\}\rangle$

a) Least refined progression K_1.

$\langle 1, \lambda\langle\mu\rangle \,.\, \{\sigma \mid 0 \leq \sigma(x) < \sigma(n) \text{ and } \sigma(x)\%2 = \mu\%2\}\rangle$

  `while` $x < n$ `do`

    $\langle 2, \lambda\langle\mu_1\mu_2\rangle \,.\, \{\sigma \mid 2\mu_2 \leq \sigma(x) < \sigma(n) \text{ and } \sigma(x)\%2 = \mu_1\%2\}\rangle$

      $x := x + 2$

    $\langle 3, \lambda\langle\mu_1\mu_2\rangle \,.\, \{\sigma \mid 2\mu_2 \leq \sigma(x) - 2 < \sigma(n) \text{ and } \sigma(x)\%2 = \mu_1\%2\}\rangle$

  `endwhile`

$\langle 4, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \sigma(n) + (\mu\%2)\}\rangle$

b) More refined progression K_2.

$\langle 1, \lambda\langle\mu\rangle \,.\, \{\sigma \mid 0 \leq \sigma(x) < \sigma(n) \text{ and } \sigma(x) = \mu\}\rangle$

  `while` $x < n$ `do`

    $\langle 2, \lambda\langle\mu_1\mu_2\rangle \,.\, \{\sigma \mid \sigma(x) < \sigma(n) \text{ and } \sigma(x) = \mu_1 + \mu_2\}\rangle$

      $x := x + 2$

    $\langle 3, \lambda\langle\mu_1\mu_2\rangle \,.\, \{\sigma \mid \sigma(x) - 2 < \sigma(n) \text{ and } \sigma(x) = \mu_1 + \mu_2 + 2\}\rangle$

  `endwhile`

$\langle 4, \lambda\langle\mu\rangle \,.\, \{\sigma \mid \sigma(x) = \sigma(n) + (\mu\%2)\}\rangle$

c) Most refined progression K_3.

Figure 11.2: Three Progressions of a Program

other words, configuration K_2 does not "compute" more environments, it just tracks the same environments in a more precise way, leading to more refined information being inferred. For example, from progression K_2 it could be inferred that the parity of x doesn't change throughout the entire execution of the loop. However, information about the sequence of values of x are still impossible to infer from K_2. In progression K_3, we distinguish between the initial values of x by dividing up the set of start environments into slices such that each slice contains environments whose value for x is equal to the slice index. In this case, it can be inferred not only that the parity of x doesn't change, but also that the values of x inside the loop are increasing. This example goes to show that the progressive semantics has an in-built mechanism for adjusting the level of abstraction of the sequencing of the states. However, this abstraction mechanism is not powerful enough. What we need to add is a mechanism for approximating slices of indexed sets, in a way that is more flexible than the superset relationship employed by Hoare logic. The main flaw of collecting-semantics-based reasoning methods is that they provide a superset AS of the semantics as its approximation. The given superset acts as a upper bound of the semantics, entailing that the collecting semantics could be any subset of AS, including the empty set. The range of possible values for the collecting semantics is the entire 2^{AS}, which is good enough for verifying safety properties, for instance, but too large to allow the inference of liveness or progress properties. To be able to handle such properties, we need a more flexible way to approximate sets. In particular, we would like to be able to infer from the approximation of a set S that the empty set is not a possible value of S. A simple means of achieving that is to provide a *subset* F of 2^{AS} as the approximation of S. The set of sets F is called a *family*, and provides a more flexible way to express a range of possible values for the set S. Since a possible choice for F is 2^{AS}, the classic collecting-

semantics-based means of abstraction becomes a special case of the approximation mechanism that we introduce. In what follows, we shall define the refinement and regression relations between progressive configurations, family configurations, and coverage and approximation relations between family configurations and progressive configurations.

Given two indexed sets Ψ and Ψ', we say that Ψ is a *refinement* of Ψ' (or Ψ' is the *regression* of Ψ) if either $\Psi \subseteq \Psi'$, or if *collect*(Ψ) is a refinement of Ψ'. An example of refinement can be seen in Figure 11.2. $K_2|_2$ is a refinement of $K_2|_1$, while $K_2|_3$ is a refinement of $K_2|_2$.

Given a progressive configuration K, we denote by *collect*(K) the progressive configuration K' such that $|K'| = |K|$ and $K'|_l = \mathit{collect}(K|_l)$ for all labels $l \in \mathit{labels}(K)$. Given two progressive configurations K and K', we say that K is a *refinement* of K' if either $K \preccurlyeq K'$, or if there exists a configuration K'' such that $K'' = \mathit{collect}(K)$ and K'' is a refinement of K'. In Figure 11.2, K_2 is a refinement of K_1, while K_3 is a refinement of K_2.

A *a family* is a set of sets of environments. An indexed family is a mapping from indices to families. We denote the set of all indexed families by **Fam**. We denote indexed families by the Greek letter Φ, possibly subscripted. Given an indexed family Φ, we denote by $\lfloor\Phi\rfloor$ the set $\{\Psi \mid \Psi(\tilde{\mu}) \in \Phi(\tilde{\mu}) \text{ for all } \tilde{\mu} \in \mathbf{Idx}\}$. Given a set of indexed sets Π, we denote by $\lceil\Pi\rceil$ the family Φ defined by $\Phi(\tilde{\mu}) = \{\Psi(\tilde{\mu}) \mid \Psi \in \Pi\}$, $\tilde{\mu} \in \mathbf{Idx}$. We also extend the $\subseteq$ operator to indexed families. Given two indexed families Φ and Φ', we say that $\Phi \subseteq \Phi'$ if for all indices $\tilde{\mu}$, $\Phi(\tilde{\mu}) \subseteq \Phi'(\tilde{\mu})$.

A configuration whose annotations are indexed families is called a *family configuration*. The set of family configurations is $\mathbf{AProg}(\mathbf{Labels} \times \mathbf{Fam})$. We denote family configurations by $\hat{K}$, possibly subscripted.

A family configuration $\hat{K}$ can be interpreted as the set of progressive config-

urations $\lfloor \hat{K} \rfloor = \{K \,|\, |K| = |\hat{K}| \text{ and } K|_l \in \lfloor \hat{K}|_l \rfloor, \text{ for all } l \in \mathit{labels}(\hat{K})\}$. Family configurations are intended to approximate the progressions; whenever a family configuration $\hat{K}$ approximates the progression of a program, we have that the progression is a member of $\lfloor \hat{K} \rfloor$.

Given two indexed families Φ and Φ', we say that Φ is a *refinement* of Φ' if either $\Phi \subseteq \Phi'$, or if $\mathit{collect}(\Phi)$ is a refinement of Φ'.

Given two family configurations $\hat{K}$ and $\hat{K}'$, we say that $\hat{K}$ is a refinement of $\hat{K}'$ if $|\hat{K}| = |\hat{K}'|$ and $\hat{K}|_l$ is a refinement of $\hat{K}'|_l$ for all labels $l \in \mathit{labels}(\hat{K})$. A family configuration $\hat{K}$ *covers* a progressive configuration K *at label* l, where $l \in \mathit{labels}(K)$, if $|\hat{K}| = |K|$, and $K|_l \in \lfloor \hat{K}|_l \rfloor$ for all indices $\tilde{\mu} \in \mathbf{Idx}$. A family configuration $\hat{K}$ *covers* a progressive configuration K if $\hat{K}$ covers K at all labels $l \in \mathit{labels}(K)$. A family configuration $\hat{K}$ approximates a progressive configuration K if it covers some regression of K.

The $\subseteq$ relationship between families induces a partial order on family configurations which we denote by $\hat{\preccurlyeq}$. Formally, given two family configurations $\hat{K}_1$ and $\hat{K}_2$, we say that $\hat{K}_1 \hat{\preccurlyeq} \hat{K}_2$ if $|\hat{K}_1| = |\hat{K}_2|$ and $\hat{K}|_l \subseteq \hat{K}|_l$, for all labels $l \in \mathit{labels}(\hat{K}_1)$. Clearly, if $\hat{K}_1$ and $\hat{K}_2$ are approximations of some progression K, then $\hat{K}_1$ is a more precise approximation than $\hat{K}_2$.

11.1 Remark Let K and $\hat{K}$ be a progressive and a family configurations, respectively, such that $\hat{K}$ covers K. The following statements hold.

a) If K is the sequence configuration $K_1 \fatsemi K_2$, then there exist the family configurations $\hat{K}_1$ and $\hat{K}_2$ such that $\hat{K} = \hat{K}_1 \fatsemi \hat{K}_2$, and $\hat{K}_i$ covers K_i, for $i \in \{1, 2\}$. Conversely, if $\hat{K}_1$ covers K_1 and $\hat{K}_2$ covers K_2, then $\hat{K}_1 \fatsemi \hat{K}_2$ covers $K_1 \fatsemi K_2$.

b) If K is the `if` configuration $\langle l_s, \Psi_s \rangle$ `if` C `then` K_c `else` K_a `endif` $\langle l_f, \Psi_f \rangle$, then there exist the indexed families Φ_s, Φ_f, and family configurations $\hat{K}_c$

and $\hat{K}_a$, such that $\hat{K} = \langle l_s, \Phi_s \rangle$ `if` C `then` $\hat{K}_c$ `else` $\hat{K}_a$ `endif` $\langle l_f, \Phi_f \rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively, and $\hat{K}_c$ and $\hat{K}_a$ covering K_c and K_a, respectively. Conversely, if Φ_s and Φ_f cover Ψ_s and Ψ_f, respectively, and $\hat{K}_c$ and $\hat{K}_a$ cover K_c and K_a, respectively, then $\langle l_s, \Phi_s \rangle$ `if` C `then` $\hat{K}_c$ `else` $\hat{K}_a$ `endif` $\langle l_f, \Phi_f \rangle$ covers $\langle l_s, \Psi_s \rangle$ `if` C `then` K_c `else` K_a `endif` $\langle l_f, \Psi_f \rangle$.

c) If K is the `while` configuration $\langle l_s, \Psi_s \rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_f \rangle$, then there exist the indexed families Φ_s, Φ_f, and family configuration $\hat{K}_b$, such that $\hat{K} = \langle l_s, \Phi_s \rangle$ `while` C `do` $\hat{K}_b$ `endwhile` $\langle l_f, \Phi_f \rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively, and $\hat{K}_b$ covering K_b. Conversely, if Φ_s and Φ_f cover Ψ_s and Ψ_f, respectively, and $\hat{K}_b$ covers K_b, respectively, then $\langle l_s, \Phi_s \rangle$ `while` C `do` $\hat{K}_b$ `endwhile` $\langle l_f, \Phi_f \rangle$ covers $\langle l_s, \Psi_s \rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_f \rangle$.

□

For clarity and conciseness of examples, we shall describe indexed families using a language similar to the one we defined for indexed sets in Section 6.1. We shall represent a indexed family Φ by formulas of the form $\lambda\langle \mu_1 \cdots \mu_k \rangle \,.\, \Upsilon(\mu_1, \ldots, \mu_k)$, where $\mu_1, \ldots, \mu_k$ are index variables, and and $\Upsilon(\mu_1, \ldots, \mu_k)$ is a set expression in which the variables $\mu_1, \ldots, \mu_k$ appear free. The indexed family Φ is defined as

$$\Phi(\tilde{\mu}) = \begin{cases} \emptyset, & \text{if } \mathit{size}(\tilde{\mu}) \neq k \\ \Upsilon(\mu_1, \ldots, \mu_k), & \text{if } \mathit{size}(\tilde{\mu}) = k \text{ and } \tilde{\mu} = \mu_1 \cdots \mu_k \end{cases}$$

For example, the expression $\lambda\langle \mu \rangle \,.\, \{S \,|\, \sigma(x) = \mu \text{ for all } \sigma \in S\}$ denotes the family Φ

defined by

$$\Phi(\tilde{\mu}) = \begin{cases} \emptyset, & \text{if } \mathit{size}(\tilde{\mu}) \neq 1 \\ \{\emptyset, \{\sigma \,|\, \sigma(x) = \mu \text{ and } \sigma(z) = 0\}, & \\ \quad \{\sigma \,|\, \sigma(x) = \mu \text{ and } \sigma(z) = 1\}, \ldots\} & \text{if } \mathit{size}(\tilde{\mu}) = 1 \text{ and } \tilde{\mu} = \mu \end{cases}$$

Figures 11.1 and 11.3 show an example of approximation. The family configuration in Figure 11.1 is an approximation of the progressive configuration in Figure 11.3. The interesting aspect of this example is that while the progressive semantics captures the values of variables z and x as a function of the number of times the loop has been executed, the family configuration abstracts this information away. Even in such conditions, the family configuration is able to express the fact that program point 3 is *live* [1] (i.e. definitely reached during execution), and that variable z has at least 2 distinct values at program point 3.

11.2 The Family Progress Operator

In Section 9.1 we introduced the progressive transfer function T^p, and showed that the progressive semantics is the least fixpoint of this operator. The definition of T^p relies on the indexed set operators defined in Figure 7.3. In this section we shall lift the indexed set operators and the progressive transfer function to indexed families in a very straightforward manner. Assume $f : \mathbf{Idx}^n \mapsto \mathbf{Idx}$ is an operator over indexed sets, where $n > 0$. We can define the family lifting $\widehat{f} : \mathbf{Fam}^n \mapsto \mathbf{Fam}$ of f as

$$\widehat{f}(\Phi_1, \ldots, \Phi_n) = \lceil \{ f(\Psi_1, \ldots, \Psi_n) \,|\, \Psi_1 \in \lfloor \Phi_1 \rfloor, \ldots, \Psi_n \in \lfloor \Phi_n \rfloor \} \rceil$$

[1] The fact that program point 3 is live comes from the condition $\emptyset \notin \{\overline{\alpha}(\Psi) \,|\, \Psi \in \lfloor \Phi_3 \rfloor\}$, where Φ_3 is the family attached to program point 3 in Figure 11.1.

$$\langle 1, \lambda\langle\rangle \,.\, \left\{ \sigma \;\middle|\; \begin{array}{l} \text{there exists } \delta \in \{0,\ldots,99\} \text{ s.t.} \\ \sigma(a)[\delta]\%10 \neq 0 \text{ and } \sigma(x) = 0 \text{ and} \\ \sigma(z) = 0 \end{array} \right\} \rangle$$

`while` $x < 100$ `do`

$$\langle 2, \lambda\langle\mu\rangle \,.\, \left\{ \sigma \;\middle|\; \begin{array}{l} \text{there exists } \delta \in \{0,\ldots,99\} \text{ s.t.} \\ \sigma(a)[\delta]\%10 \neq 0 \text{ and } \sigma(x) = \mu \text{ and} \\ \sigma(z) = \sum_{\delta'=0}^{\mu-1} \sigma(a)[\delta'] \end{array} \right\} \rangle$$

$z := (z + a[x]) \,\%\, 10$

$$\langle 3, \lambda\langle\mu\rangle \,.\, \left\{ \sigma \;\middle|\; \begin{array}{l} \text{there exists } \delta \in \{0,\ldots,99\} \text{ s.t.} \\ \sigma(a)[\delta]\%10 \neq 0 \text{ and } \sigma(x) = \mu \text{ and} \\ \sigma(z) = \sum_{\delta'=0}^{\mu} \sigma(a)[\delta'] \end{array} \right\} \rangle$$

$x := x + 1$

$$\langle 4, \lambda\langle\mu\rangle \,.\, \left\{ \sigma \;\middle|\; \begin{array}{l} \text{there exists } \delta \in \{0,\ldots,99\} \text{ s.t.} \\ \sigma(a)[\delta]\%10 \neq 0 \text{ and } \sigma(x) = \mu + 1 \text{ and} \\ \sigma(z) = \sum_{\delta'=0}^{\mu} \sigma(a)[\delta'] \end{array} \right\} \rangle$$

`endwhile`

$$\langle 5, \lambda\langle\rangle \,.\, \left\{ \sigma \;\middle|\; \begin{array}{l} \text{there exists } \delta \in \{0,\ldots,99\} \text{ s.t.} \\ \sigma(a)[\delta]\%10 \neq 0 \text{ and } \sigma(x) = 100 \text{ and} \\ \sigma(z) = \sum_{\delta'=0}^{99} \sigma(a)[\delta'] \end{array} \right\} \rangle$$

Figure 11.3: Progression of Distinct Values Example

Figure 11.4 shows the definitions of the operators $\widehat{\mathit{assign}}$, $\widehat{\mathit{filter}}$, $\widehat{\mathit{before}}$, $\widehat{\mathit{collect}}$ and $\widehat{\bigcup}$, which are the liftings of the corresponding indexed set operators defined in Figure 7.3.

11.2 Remark Given the indexed sets Ψ and Ψ', and the indexed families Φ and Φ', such that Φ covers Ψ and Φ' covers Ψ', the following statements hold.

a) $\widehat{\mathit{assign}}(x, E, \Phi)$ covers $\mathit{assign}(x, E, \Psi)$, for all program variables x and program expressions E.

b) $\widehat{\mathit{filter}}(C, \Phi)$ covers $\mathit{filter}(C, \Psi)$, for all program constraints C.

c) $\widehat{\mathit{before}}(\Phi, \Phi')$ covers $\mathit{before}(\Psi, \Psi')$.

$\widehat{assign} : \mathbf{Var} \times \mathbf{Expr} \times \mathbf{Fam} \mapsto \mathbf{Fam}$

$\widehat{assign}(x, E, \Phi) = \lceil \{ assign(x, E, \Psi) \mid \Psi \in \lfloor \Phi \rfloor \} \rceil$

$\widehat{filter} : \mathbf{Constr} \times \mathbf{Fam} \mapsto \mathbf{Fam}$

$\widehat{filter}(C, \Phi) = \lceil \{ filter(C, \Psi) \mid \Psi \in \lfloor \Phi \rfloor \} \rceil$

$\widehat{\cup} : \mathbf{Fam} \times \mathbf{Fam} \mapsto \mathbf{Fam}$

$\Phi \mathbin{\widehat{\cup}} \Phi' = \lceil \{ \Psi \cup \Psi' \mid \Psi \in \lfloor \Phi \rfloor \text{ and } \Psi' \in \lfloor \Phi' \rfloor \} \rceil$

$\widehat{before} : \mathbf{Fam} \times \mathbf{Fam} \mapsto \mathbf{Fam}$

$\widehat{before}(\Phi, \Phi') = \lceil \{ before(\Psi, \Psi') \mid \Psi \in \lfloor \Phi \rfloor \text{ and } \Psi' \in \lfloor \Phi' \rfloor \} \rceil$

$\widehat{collect} : \mathbf{Fam} \mapsto \mathbf{Fam}$

$\widehat{collect}(\Phi) = \lceil \{ collect(\Psi) \mid \Psi \in \lfloor \Phi \rfloor \} \rceil$

Figure 11.4: Family Operators

d) $\widehat{collect}(\Phi)$ covers $collect(\Psi)$.

e) $\Phi \mathbin{\widehat{\cup}} \Phi'$ covers $\Psi \cup \Psi'$.

□

Using the operators defined in Figure 7.3, we define the *family transfer function* $\widehat{T}$ in Figure 11.5, as the lifting of the progressive transfer function T^p to family configurations.

The $\widehat{T}$ operator can be used both as a means to check whether a family configuration covers the progression of a program, and as a *refinement operator*, in the sense that if $\hat{K}$ covers the progression of a program, then $\widehat{T}(\hat{K})$ is a *more precise* cover, that is, $\widehat{T}(\hat{K}) \hat{\preccurlyeq} \hat{K}$.

One important property that the family transfer function $\widehat{T}$ inherits from the

$$\widehat{T}\left(\begin{array}{l}\langle l_s, \Phi_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \Phi_f\rangle\end{array}\right) \quad = \quad \begin{array}{l}\langle l_s, \Phi_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \Phi_s\rangle\end{array}$$

$$\widehat{T}\left(\begin{array}{l}\langle l_s, \Phi_s\rangle \\ \quad x := E \\ \langle l_f, \Phi_f\rangle\end{array}\right) \quad = \quad \begin{array}{l}\langle l_s, \Phi_s\rangle \\ \quad x := E \\ \langle l_f, \widehat{\mathit{assign}}(x, e, \Phi_s)\rangle\end{array}$$

$$\widehat{T}\left(\begin{array}{l}\langle l_s, \Phi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \qquad \hat{K}_1 \\ \quad \texttt{else} \\ \qquad \hat{K}_2 \\ \texttt{endif} \\ \langle l_f, \Phi_f\rangle\end{array}\right) \quad = \quad \begin{array}{l}\langle l_s, \Phi_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \qquad \langle l_{cs}, \widehat{\mathit{filter}}(C, \hat{K}_c|_{l_{cs}})\rangle \fatsemi \widehat{T}(\hat{K}_c) \\ \quad \texttt{else} \\ \qquad \langle l_{as}, \widehat{\mathit{filter}}(\neg C, \hat{K}_a|_{l_{as}})\rangle \fatsemi \widehat{T}(\hat{K}_a) \\ \texttt{endif} \\ \langle l_f, \hat{K}_c|_{l_{cf}} \ \widehat{\cup} \ \hat{K}_a|_{l_{af}}\rangle\end{array}$$

where $l_{is} = \mathit{first}(\hat{K}_i),\ l_{if} = \mathit{last}(\hat{K}_i),\ i \in \{1, 2\}$

$$\widehat{T}\left(\begin{array}{l}\langle l_s, \Phi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \hat{K}_b \\ \texttt{endwhile} \\ \langle l_f, \Phi_f\rangle\end{array}\right) \quad = \quad \begin{array}{l}\langle l_s, \Phi_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \langle l'_s, \widehat{\mathit{filter}}(C, \widehat{\mathit{before}}(\Phi_s, \hat{K}|_{l'_f}))\rangle \fatsemi \widehat{T}(\hat{K}) \\ \texttt{endwhile} \\ \langle l_f, \widehat{\mathit{filter}}(\neg C, \Phi_s \ \widehat{\cup} \ \widehat{\mathit{collect}}(\hat{K}|_{l'_f}))\rangle\end{array}$$

where $l_s = \mathit{first}(\hat{K}),\ l_f = \mathit{last}(\hat{K})$

$$\widehat{T}\left(\hat{K}_1 \fatsemi \hat{K}_2\right) \quad = \quad \widehat{T}\left(\hat{K}_1\right) \fatsemi \widehat{T}\left(\hat{K}_2\right)$$

Figure 11.5: Progressive Transfer Function for Family Configurations

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \,|\, 0 \leq \sigma(x) < 2\}\rangle$
`if` $x\%2 = 1$ `then`
$\langle 2, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 1\}\rangle$
$x := x + 1$
$\langle 3, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 2\}\rangle$
`else`
$\langle 4, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 0\}\rangle$
`skip`
$\langle 5, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 0\}\rangle$
`endif`
$\langle 6, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 0 \text{ or } \sigma(x) = 2\}\rangle$

$\langle 1, \lambda\langle\rangle \,.\, \{\sigma \,|\, 10 \leq \sigma(x) < 12\}\rangle$
`if` $x\%2 = 1$ `then`
$\langle 2, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 11\}\rangle$
$x := x + 1$
$\langle 3, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 12\}\rangle$
`else`
$\langle 4, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 10\}\rangle$
`skip`
$\langle 5, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 10\}\rangle$
`endif`
$\langle 6, \lambda\langle\rangle \,.\, \{\sigma \,|\, \sigma(x) = 10 \text{ or } \sigma(x) = 12\}\rangle$

(a) Two progressions

$\langle 1, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, 0 \leq \sigma(x) < 2\}, \{\sigma \,|\, 10 \leq \sigma(x) < 12\}\}\rangle$
`if` $x\%2 = 1$ `then`
$\langle 2, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 1\}, \{\sigma \,|\, \sigma(x) = 11\}\}\rangle$
$x := x + 1$
$\langle 3, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 2\}, \{\sigma \,|\, \sigma(x) = 12\}\}\rangle$
`else`
$\langle 4, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 0\}, \{\sigma \,|\, \sigma(x) = 10\}\}\rangle$
`skip`
$\langle 5, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 0\}, \{\sigma \,|\, \sigma(x) = 10\}\}\rangle$
`endif`
$\langle 6, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 0 \text{ or } \sigma(x) = 2\}, \{\sigma \,|\, \sigma(x) = 10 \text{ or } \sigma(x) = 12\}\}\rangle$

(b) Family configuration that best approximates the two progressions

Figure 11.6: Two Progressions and Their Best Approximation

$\langle 1, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, 0 \leq \sigma(x) < 2\}, \{\sigma \,|\, 10 \leq \sigma(x) < 12\}\}\rangle$
  `if` $x\%2 = 1$ `then`
    $\langle 2, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 1\}, \{\sigma \,|\, \sigma(x) = 11\}\}\rangle$
      $x := x + 1$
    $\langle 3, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 2\}, \{\sigma \,|\, \sigma(x) = 12\}\}\rangle$
  `else`
    $\langle 4, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 0\}, \{\sigma \,|\, \sigma(x) = 10\}\}\rangle$
      `skip`
    $\langle 5, \lambda\langle\rangle \,.\, \{\{\sigma \,|\, \sigma(x) = 0\}, \{\sigma \,|\, \sigma(x) = 10\}\}\rangle$
  `endif`
$\langle 6, \lambda.\langle\rangle\{\{\sigma \,|\, \sigma(x) = 0 \text{ or } \sigma(x) = 2\}, \{\sigma \,|\, \sigma(x) = 10 \text{ or } \sigma(x) = 12\},$
$\{\sigma \,|\, \sigma(x) = 0 \text{ or } \sigma(x) = 12\}, \{\sigma \,|\, \sigma(x) = 10 \text{ or } \sigma(x) = 2\}\}\rangle$

(a) Straightforward approximation of the Progression in Figure 11.6.

$\langle 1, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, 10\mu \leq \sigma(x) < 10\mu + 2, \text{ for all } \sigma \in \Sigma\}\rangle$
  `if` $x\%2 = 1$ `then`
    $\langle 2, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, \sigma(x) = 10\mu + 1, \text{ for all } \sigma \in \Sigma\}\rangle$
      $x := x + 1$
    $\langle 3, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, \sigma(x) = 10\mu + 2, \text{ for all } \sigma \in \Sigma\}\rangle$
  `else`
    $\langle 4, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, \sigma(x) = 10\mu, \text{ for all } \sigma \in \Sigma\}\rangle$
      `skip`
    $\langle 5, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, \sigma(x) = 10\mu, \text{ for all } \sigma \in \Sigma\}\rangle$
  `endif`
$\langle 6, \lambda\langle\mu\rangle \,.\, \{\Sigma \,|\, \sigma(x) = 10\mu \text{ or } \sigma(x) = 10\mu + 2, \text{ for all } \sigma \in \Sigma\}\rangle$

(b) Refined approximation of the Progression in Figure 11.6.

Figure 11.7: Fixpoints of $\widehat{T}$

progressive transfer function is the uniqueness of its fixpoint, for a fixed family attached to the first program point. The reason for this property is very much the same as for the uniqueness of the fixpoint of T^p, and it can be intuitively explained in a manner similar to the discussion given in Section 9.3. This property is also captured by the following proposition.

11.3 Proposition Let K be a progressive configuration, and $\hat{K}$ a family configuration that covers K. Then, $\widehat{T}(\hat{K})$ covers $T^p(K)$.

Proof: Relegated to Appendix A, on page 310.

The $\widehat{T}$ operator resembles the *strongest postcondition propagation* employed by Hoare-like reasoning. However, strongest postcondition propagation has a major shortcoming, residing in the fact that the classical collective-semantics-based transfer function T may have more than just one fixed point. Strongest postcondition propagation is thus limited to being as precise as the greatest fixpoint of T. Since the $\widehat{T}$ operator has a unique fixpoint, one may think that it could be possible to compute an arbitrarily precise family-configuration-based approximation of the progressive semantics. However, that is not true. The $\widehat{T}$ operator has limitations of a different nature, which will be explained in what follows. Consider the two progressions given in Figure 11.6a. They are progressions of the same program, but w.r.t. two different sets of start environments. Figure 11.6b represents the desired family configuration that covers both progressions of Figure 11.6a. This family configuration contains only sets of states that appear in some of the progressions of Figure 11.6a, that is, it contains no "garbage". However, the family configuration given in Figure 11.6b is not a fixpoint of $\widehat{T}$. Figure 11.7a shows a family configuration with the same annotation for the first program point as in Figure 11.6b, and which is a fixpoint of $\widehat{T}$. We note the fact that this family configuration is less precise than the ideal one,

by having two sets of environments at the last program point that do not appear in any of the progressions given in Figure 11.6a. These sets of environments are $\{\sigma \mid \sigma(x) = 0 \text{ or } \sigma(x) = 12\}$ and $\{\sigma \mid \sigma(x) = 10 \text{ or } \sigma(x) = 2\}$. They were added by the operator $\widehat{\cup}$, which joins the information coming out of the two branches of the `if` statement.

To overcome this flaw, we can refine the family attached to the first program point, and compute the fixpoint of $\widehat{T}$ using the newly refined family. Figure 11.7b shows the result of this process. We note that the "refined" family does not contain any "garbage".

11.3 A Sufficient Condition for Coverage

In this section we prove that a family configuration covers the progression of a program if it covers it at the first program point and if the family configuration is a post-fixpoint of the family progress operator. The proof is by induction on the structure of the configuration, and has been split into several parts for readability. The main result of this section is given by Theorem 11.13. Propositions 11.4, 11.5 and 11.6 show that if a family configuration is a postfixpoint of $\widehat{T}$, then so are its sub-configurations. This allows the use of the induction hypothesis on the sub-configurations. Propositions 11.7, 11.8, 11.9, 11.12, and Remarks 11.10, and 11.11 show that whenever the family sub-configurations cover the progressive sub-configurations, the same relationship holds between the family and the progressive configurations. All these partial results are combined to produce the proof of Theorem 11.13.

We start by defining the operators *seq* and *extr* for families and family configurations. The operator *seq* is useful in coalescing a (possibly infinite) sequence of

families coming from repeated runs through the body of a `while` loop into one family. The *extr* operator performs the dual operation, that is, extracts a configuration that performs one run through the body of a `while` loop. These operators can be extended to family configurations in a straightforward manner and will be useful in applying an induction hypothesis to `while` programs.

Formally, given a (possibly infinite) sequence of families $\Phi_1, \Phi_2, \ldots, \Phi_n, \ldots$, we denote by $seq(\Phi_1, \Phi_2, \ldots, \Phi_n, \ldots)$ the family Φ defined by

$$\Phi(\tilde{\mu}) = \begin{cases} \emptyset, & \text{if } \tilde{\mu} = \epsilon \\ \Phi_i(\tilde{\mu}') & \text{if } \tilde{\mu} = i\tilde{\mu}' \end{cases}$$

Also, given a family Φ, and a natural number i, we denote by $extr(\Phi, i)$ the family Φ' such that $\Phi'(\tilde{\mu}) = \Phi(i\tilde{\mu})$, for all $\mu \in \mathbf{Idx}$. The operators *seq* and *extr* can be extended to family configurations in the following way. Given a (possibly infinite) sequence of family configurations $\hat{K}_1, \hat{K}_2, \ldots, \hat{K}_n, \ldots$, such that $|\hat{K}_1| = |\hat{K}_2| = \cdots = |\hat{K}_n| = \cdots$, we denote by $seq(\hat{K}_1, \hat{K}_2, \ldots, \hat{K}_n, \ldots)$ the family configuration $\hat{K}$ such that $|\hat{K}| = |\hat{K}_1|$ and $\hat{K}|_l = seq(\hat{K}_1|_l, \hat{K}_2|_l, \ldots, \hat{K}_n|_l, \ldots)$, for all labels $l \in labels(\hat{K}_1)$. Similarly, given a family configuration $\hat{K}$, and a natural number i, we denote by $extr(\hat{K}, i)$ the family configuration $\hat{K}'$, such that $|\hat{K}'| = |\hat{K}|$ and $\hat{K}'|_l = extr(\hat{K}|_l, i)$, for all labels $l \in labels(\hat{K})$.

11.4 Proposition Let P be the sequence program $P_1 \fatsemi P_2$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_1$ and $\hat{K}_2$ the sub-configurations of $\vec{K}$ such that $|\hat{K}_1| = P_1$ and $|\hat{K}_2| = P_2$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_1) \preccurlyeq \hat{K}_1$ and $\widehat{T}(\hat{K}_2) \preccurlyeq \hat{K}_2$.

Proof: Relegated to Appendix A, on page 311.

11.5 Proposition Let P be the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_c$ and $\hat{K}_a$ the

sub-configurations of $\hat{K}$ such that $|\hat{K}_a| = P_a$ and $|\hat{K}_c| = P_c$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_a) \preccurlyeq \hat{K}_a$ and $\widehat{T}(\hat{K}_c) \preccurlyeq \hat{K}_c$.

Proof: Relegated to Appendix A, on page 312.

11.6 Proposition Let P be the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_b$ the sub-configuration of $\hat{K}$ such that $|\hat{K}_b| = P_b$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_b) \preccurlyeq \hat{K}_b$.

Proof: Relegated to Appendix A, on page 312.

11.7 Proposition Assume P is the skip program $\langle l_s \rangle$ `skip` $\langle l_f \rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration that covers K at l_s. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\hat{K}$ covers K at l_f.

Proof: Relegated to Appendix A, on page 312.

11.8 Proposition Assume P is the skip program $\langle l_s \rangle\, x := E\, \langle l_f \rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration that covers K at l_s. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\hat{K}$ covers K at l_f.

Proof: Relegated to Appendix A, on page 313.

11.9 Proposition Assume P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration such that $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$. The following statements hold.

a) If $\hat{K}$ covers K at l_s, then $\hat{K}$ covers K at *first*(P_c) and *first*(P_a).

b) If $\hat{K}$ covers K at *last*(P_c) and *last*(P_a), then $\hat{K}$ covers K at l_f.

Proof: Relegated to Appendix A, on page 313.

11.10 Remark Given a progression K and a family configuration $\hat{K}$, we have that $\hat{K}$ covers K if and only if $extr(\hat{K}, \mu)$ covers $extr(K, \mu)$ for all $\mu \in \mathbb{N}$. Similarly, given two family configurations $\hat{K}_1$ and $\hat{K}_2$, we have that $\hat{K}_1 \preccurlyeq \hat{K}_2$ if and only if $extr(\hat{K}_1, \mu) \preccurlyeq extr(\hat{K}_2, \mu)$ for all $\mu \in \mathbb{N}$. □

11.11 Remark The following statements hold.

a) $\widehat{T}(seq(\hat{K}_1, \ldots, \hat{K}_n)) = seq(\widehat{T}(\hat{K}_1), \ldots, \widehat{T}(\hat{K}_n))$.

b) $\widehat{T}(extr(\hat{K}, \mu)) = extr(\widehat{T}(\hat{K}), \mu)$, for all $\mu \in \mathbb{N}$.

c) $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$ if and only if $\widehat{T}(extr(\hat{K}, \mu)) \preccurlyeq extr(\hat{K}, \mu)$, for all $\mu \in \mathbb{N}$.

□

11.12 Proposition Assume P is the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration such that $\hat{K}$ covers K at l_s, and $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$. The following statements hold.

a) $extr(\hat{K}|_{first(P_b)}, 0)$ covers $extr(K|_{first(P_b)}, 0)$.

b) $extr(\hat{K}|_{last(P_b)}, \mu) \preccurlyeq extr(\hat{K}|_{first(P_b)}, \mu + 1)$.

c) If $\hat{K}|_{last(P_b)}$ covers $K|_{last(P_b)}$, then $\hat{K}|_{l_f}$ covers $K|_{l_f}$.

Proof: Relegated to Appendix A, on page 314.

The following theorem is the main result of this section. It proves that a family configuration covers the progression of a program if it covers it at the first program

point and if the family configuration is a post-fixpoint of the family progress operator. The result is significant in the sense that it induces a similar result for symbolic configurations in Chapter 14, which in turn leads to a proof method for progressive properties of programs.

11.13 Theorem Let P be a program and Σ_0 a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration that covers K at $\mathit{first}(K)$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\hat{K}$ covers K.

Proof: The proof is by structural induction on P. Propositions 11.7 and 11.8 take care of the cases when P is either a `skip` statement or an assignment. Assume now that P is the sequence program $P_1 \fatsemi P_2$. Then, we have that $K = K_1 \fatsemi K_2$ and $\hat{K} = \hat{K}_1 \fatsemi \hat{K}_2$, where $|K_1| = |\hat{K}_1| = P_1$ and $|K_2| = |\hat{K}_2| = P_2$. According to the induction hypothesis, $\hat{K}_1$ covers K_1. This entails that $\hat{K}_1|_{\mathit{last}(P_1)}$ covers $K_1|_{\mathit{last}(P_1)}$. Now, we have that $\hat{K}_1|_{\mathit{last}(P_1)} = \hat{K}_2|_{\mathit{first}(P_2)}$ and $K_1|_{\mathit{last}(P_1)} = K_2|_{\mathit{first}(P_2)}$. Applying the induction hypothesis again, it follows that $\hat{K}_2$ covers K_2. Since $\hat{K}|_l$ covers $K|_l$ for all labels $l \in \mathit{labels}(P)$, it follows that $\hat{K}$ covers K. Assume now that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. Then we have that $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0 \rangle$ `if` C `then` K_c `else` K_a `endif` $\langle l_f, \Psi_f \rangle$, and $\hat{K} = \langle l_s, \Phi_s \rangle$ `if` C `then` $\hat{K}_c$ `else` $\hat{K}_a$ `endif` $\langle l_f, \Phi_f \rangle$, where $|K_c| = |\hat{K}_c| = P_c$, $|K_a| = |\hat{K}_a| = P_a$, Ψ_f is an indexed set, and Φ_s and Φ_f are indexed families. According to Proposition 11.9 $\hat{K}_c|_{\mathit{first}(P_c)}$ covers $K_c|_{\mathit{first}(P_c)}$, and $\hat{K}_a|_{\mathit{first}(P_a)}$ covers $K_a|_{\mathit{first}(P_a)}$. Using the inductive hypothesis, it follows that $\hat{K}_c$ covers K_c, and $\hat{K}_a$ covers K_a. This entails that $\hat{K}$ covers K at $\mathit{last}(P_c)$ and $\mathit{last}(P_a)$, and by using Proposition 11.9 again, it follows that $\hat{K}$ covers K at l_f. As a result, $\hat{K}$ covers K. Assume now that P is the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$. Then, we have that $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0 \rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_s \rangle$, and

$\hat{K} = \langle l_s, \Phi_s \rangle$ `while` C `do` $\hat{K}_b$ `endwhile` $\langle l_f, \Phi_s \rangle$, $|K_b| = |\hat{K}_b| = P_b$, Ψ_f is an indexed set, and Φ_s and Φ_f are indexed families. We need to prove two things.

a) $\hat{K}_b$ covers K_b.

b) $\hat{K}|_{l_f}$ covers $K|_{l_f}$.

Using Remark 11.10 we shall prove a) by proving by induction that $\mathit{extr}(\hat{K}, \mu)$ covers $\mathit{extr}(K, \mu)$. The base case follows from Proposition 11.12 and the induction hypothesis. For the induction case, assume that the statement holds for μ. Then, from Proposition 11.12 it follows that $\mathit{extr}(\hat{K}, \mu + 1)|_{\mathit{first}(P_b)}$ covers $\mathit{extr}(K, \mu + 1)|_{\mathit{first}(P_b)}$. Applying the induction hypothesis again, it follows that $\mathit{extr}(\hat{K}, \mu + 1)$ covers $\mathit{extr}(K, \mu + 1)$. Statement b) follows from a) by applying Proposition 11.12 again. □

11.4 Discussion

In this chapter we have established that indexed families, as means of abstracting indexed sets, are capable of capturing an abstraction of the sequence of environments that occur at a program point, as well as specifying in a flexible way a range of values for the projection of the collecting semantics on a program point. In order to better understand these features, let us consider the following indexed family, which might be an approximation of a progression at a program point inside a `while` loop.

$$\Phi = \lambda\langle\mu\rangle \,.\, \{\{\sigma\} \,|\, \sigma(x) \in \{\mathit{prime}(\delta) \,|\, \mu \leq \delta \leq \mu + 2\}\}$$

where $\mathit{prime}(i)$ is a mapping that returns the i^{th} prime number. The variable μ, which ranges over natural numbers, models the passage of time, in the sense that environments corresponding to larger values of μ may appear later during the

execution of the program. From the indexed family given above it can be inferred that, while the prime numbers may not appear in increasing order, the prime number 2 will definitely occur before the prime number 7, if it is the case that both 2 and 7 would occur at all during the execution of the program.

Moreover, the expression $collect(\Phi)$, which evaluates to

$$\Phi' = \lambda\langle\rangle\,.\,\{S \,|\, \text{for all numbers } n,\ \text{there exists } \sigma \in S \text{ s.t. } \sigma(x) \in \{prime(\delta) \,|\, \mu \leq \delta \leq \mu + 2\}\}$$

is an approximation for the collecting semantics of the program projected on the label at hand. From Φ', it can be inferred that this projection is not empty, that is, the program point at hand is *live*.

It may appear that progressive semantics and family-configuration-based approximations abstract away the sequencing between events occurring at different program points. It is however possible to infer such sequencing using the progressive index of that event, together with the program label ordering introduced in Section 7.2. Indeed, if we have a family configurations $\hat{K}$, two program labels, l_1 and l_2, and two progressive indices $\tilde{\mu}_1$ and $\tilde{\mu}_2$, we can say that the property represented by the family $\hat{K}|_{l_1}(\tilde{\mu}_1)$ occurs before the property of the family $\hat{K}|_{l_2}(\tilde{\mu}_2)$ if $\tilde{\mu}_1 \leq \tilde{\mu}_2$, or if $\tilde{\mu}_1 = \tilde{\mu}_2$ and $l_1 \circledcirc l_2$.

In the rest of this thesis we shall develop means of program reasoning that would allow the inference and verification of progressive properties. We shall start by defining family description languages, and based on that, a Hoare-like calculus and a strongest postcondition propagation operator. Under these circumstances, it is important to establish that the family transfer function has potential for refining progressive information. The following lemmas contribute towards that goal, by showing that repeated applications of the family progress operator on the cover of a progression K would lead to more precise covers of K.

11.14 Lemma Let P be a program, and K its progression w.r.t. some set of start environments Σ_0. Assume that the family configuration $\hat{K}$ covers K. Then, $\widehat{T}(\hat{K})$ covers K as well.

Proof: The proof follows immediately from Theorem 11.3. Indeed, since $\hat{K}$ covers K, it follows that $\widehat{T}(\hat{K})$ covers $T^p(K)$. Since K is a progression of P, we have that $K = T^p(K)$, which entails that $\widehat{T}(\hat{K})$ covers K. □

11.15 Lemma Let P be a program and Σ_0 a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration such that $\hat{K}$ covers K at *first*(K) and $\widehat{T}(\hat{K}) \hat{\preccurlyeq} \hat{K}$. Then, $\widehat{T}^n(\hat{K})$ covers K, for all $n \geq 0$.

Proof: The proof is by induction on n. The lemma is obviously true for $n = 0$. For $n > 0$, we assume that $\widehat{T}^{n-1}(\hat{K})$ covers K. According to Lemma 11.14, $\widehat{T}(\widehat{T}^{n-1}(\hat{K}))$ covers K as well, and this proves the induction case. □

Lemma 11.15 induces a mechanism for refining the cover $\hat{K}$ of a progression K. Indeed, once we have verified that $\widehat{T}(\hat{K}) \hat{\preccurlyeq} \hat{K}$, we can produce a more precise cover by computing $\widehat{T}^n(\hat{K})$, for some $n > 0$.

Chapter 12

Progressive Hoare Calculi

In the previous chapter we argued in favor of the benefits of approximating the progressive semantics of a program using family configurations. In order to progress towards a progressive reasoning framework, we shall introduce a "progressive" Hoare-like calculus. Such a calculus would allow us to reason formally about programs and explore the possibility of automating this task. In this chapter we start with defining family description languages as sets of formulas that are interpreted as indexed families. We continue with defining a progressive Hoare-style calculus whose Hoare triples are made up of programs sandwiched between family description language formulas. We show that our progressive Hoare-style calculus is correct, and then we introduce two concrete family description languages and investigate the progressive Hoare calculi based on these two languages.

12.1 Hoare-Style Calculi

A *family description language* (FDL) is a language whose formulas are interpreted as indexed families. Given a family description language L, and a formula $\mathcal{F} \in \mathcal{L}$, we denote by $[\![\mathcal{F}]\!]$ the interpretation of $\mathcal{F}$. Since the manipulation of such formulas has

to mimic the manipulation of families, it is also useful to define the meta-operators *Assign*, *Filter*, *Before*, *Collect*, $\sqcup$, and *Seq*, which correspond to the $\widehat{assign}$, $\widehat{filter}$, $\widehat{before}$, $\widehat{collect}$, $\widehat{\cup}$ and *seq* operators, respectively. More specifically, given a family description language $\mathcal{L}$, two families $\mathcal{F}, \mathcal{F}' \in \mathcal{L}$, a program variable x, a program expression E, and a program constraint C, we assume that the following expressions can be translated into formulas of $\mathcal{L}$: $Assign(x, E, \mathcal{F})$, $Filter(C, \mathcal{F})$, $Before(\mathcal{F}, \mathcal{F}')$, $Collect(\mathcal{F})$, and $\mathcal{F} \sqcup \mathcal{F}'$.

We also assume that, based on a family description language $\mathcal{L}$, we can build a formal system $\mathcal{L}^*$, that can be used to establish the validity of formulas of the form $\mathcal{F} \vdash \mathcal{F}'$, where $\mathcal{F}, \mathcal{F}' \in \mathcal{L}$ and $\vdash$ is a symbol that is not part of $\mathcal{L}$. Whenever we have two formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ such that both $\mathcal{F}_1 \vdash \mathcal{F}_2$ and $\mathcal{F}_2 \vdash \mathcal{F}_1$ are valid in $\mathcal{L}^*$, we write $\mathcal{F}_1 \equiv \mathcal{F}_2$.

12.1 Definition We say that a family description language $\mathcal{L}$ and its associated formal system $\mathcal{L}^*$ are *well-defined* if the following statements hold:

a) $\widehat{assign}(x, E, [\![\mathcal{F}]\!]) \subseteq [\![Assign(x, E, \mathcal{F})]\!]$.

b) $\widehat{filter}(C, [\![\mathcal{F}]\!]) \subseteq [\![Filter(C, \mathcal{F})]\!]$.

c) $\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]) \subseteq [\![Before(\mathcal{F}_1, \mathcal{F}_2)]\!]$.

d) $\widehat{collect}([\![\mathcal{F}]\!]) \subseteq [\![Collect(\mathcal{F})]\!]$.

e) $[\![\mathcal{F}_1]\!] \,\widehat{\cup}\, [\![\mathcal{F}_2]\!] \subseteq [\![F_1 \sqcup F_2]\!]$.

f) $seq([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!], \ldots) \subseteq [\![Seq(\mathcal{F}_1, \mathcal{F}_2, \ldots)]\!]$.

g) $\mathcal{F} \vdash \mathcal{F}'$ is a theorem in $\mathcal{L}^*$ only if $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$.

where $\mathcal{F}, \mathcal{F}', \mathcal{F}_1, \mathcal{F}_2, \ldots \in \mathcal{L}$, x is a program variable, E is a program expression, and C is a program constraint. $\square$

For the sole reason of introducing a Hoare-style calculus, we define a simpler, imperative, non-annotated programming language in Figure 12.1. This language follows the syntactic structure of annotated programs defined in Section 2.1 and has the same expressive power. We shall denote non-annotated programs by $\mathcal{P}$, possibly subscripted or primed, and we shall simply refer to them as *programs* throughout the rest of this chapter. The definitions of program variables, expressions and constraints remain the same as given in Figure 4.1.

Given a program $\mathcal{P}$ and two formulas $\mathcal{F}_1, \mathcal{F}_2$ of some family description language $\mathcal{L}$, a progressive Hoare triple is a construct of the form $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$. The intended interpretation of such a triple is that if the progressive behavior before the start of the program fragment $\mathcal{P}$ is described by an indexed set Ψ_1, covered by the indexed family $[\![\mathcal{F}_1]\!]$, then the progressive behavior at the end of the program fragment $\mathcal{P}$ is described by an indexed set Ψ_2 covered by the indexed family $[\![\mathcal{F}_2]\!]$.

The *progressive Hoare calculus* $PH(\mathcal{L}^*)$, based on the formal system $\mathcal{L}^*$, is defined as the formal proof system given in Figure 12.2. The proof rules of this calculus are very similar to the classic Hoare rules presented in Section 2.1 and shall be useful in establishing the correctness of our propagation algorithm in the next chapter. The proof rules are presented in a manner similar to [AO97]. Specifically, a proof rule has the form

$$\frac{\text{set of premises}}{\text{conclusion}} \quad \text{side condition}$$

where the premises and the conclusion are progressive Hoare triples, and the side conditions are a set of meta-conditions that have to hold in order for the rule to be applied. Informally, the rules provide a means of establishing the correctness of triples concerning some program $\mathcal{P}$ on the assumption that the triples concerned with the components of $\mathcal{P}$ are correct. In what follows, we shall formalize this

process by defining the notions of correctness of a progressive Hoare triple and progressive Hoare proof. Before doing that, however, we proceed with a brief explanation of the rules in Figure 12.2. The rules (PRECOND) and (POSTCOND) represent precondition weakening and postcondition strengthening. They rely on the ability to prove the statements $\mathcal{F}_1' \vdash \mathcal{F}_1$ and $\mathcal{F}_2 \vdash \mathcal{F}_2'$ in the underlying formal system $\mathcal{L}^*$. The rule (DISJ) combines two triples into one that has the disjunction of the preconditions as a precondition, and the disjunction of the postconditions as a postcondition. The rules (SKIP) and (ASSIG) are in fact axiom schemes that produce elementary triples for skip statements and assignments. The sequencing rule (SEQ) is identical to the equivalent one in Hoare logic. Finally, the (IF) and (WHILE) rules deserve more attention. The intersection operation performed on sets of states in the collecting semantics setting is now replaced by *Filter*, and the usual disjunction is replaced by the $\sqcup$ operator which models the component-wise disjunction of indexed families. For the (WHILE) rule, we replace the union of sets of states from before the `while` loop and the bottom of the body of the `while` loop by the expression $\mathit{Before}(\mathcal{F}_1, \mathcal{F}_2)$ which keeps the slices of the indexed families in sequence. The effect of the slices at the bottom of the body of the `while` loop on the program point outside the `while` loop is modeled by the *Collect* operator. This effect must be augmented with the slices at the program point before the `while` loop, and then filtered with respect to the negation of the while condition. In the rest of this chapter we shall be concerned with proving the soundness of this calculus, and then defining two family description languages together with their associated formal systems, and investigating the concrete progressive Hoare calculi based on the two languages.

We shall now define the notions of correctness of a progressive Hoare triple and progressive Hoare proof.

12.2 Definition Let $\mathcal{P}$ be a program. A progressive Hoare triple $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is *correct* if, for all progressions K of $\mathcal{P}$ w.r.t. some set of start environments, whenever $K|_{first(K)} \in \lfloor[\![\mathcal{F}_1]\!]\rfloor$ it follows that $K|_{last(K)} \in \lfloor[\![\mathcal{F}_2]\!]\rfloor$. □

Intuitively, a progressive Hoare triple $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is correct if the interpretations of the two fromulas $\mathcal{F}_1$ and $\mathcal{F}_2$ cover the annotations of K at the first and last program point.

The definition of a proof follows the classical definition of a proof in a formal system, as a sequence of formulas, each of them following from some preceding rules in the sequence by the application of a inference rule.

12.3 Definition A *progressive Hoare proof* is a sequence of Hoare triples $H_1, H_2, \ldots, H_k$ such that for every triple H_i, there exist a set of triples $\{H_{j_1}, \ldots, H_{j_l}\}$ that appear earlier in the sequence (that is, $j_p < i$, for all p, $1 \leq p \leq l$), such that

$$\frac{H_{j_1}, \ldots, H_{j_l}}{H_i} \quad \mathcal{C}$$

is an instance of a progressive Hoare rule with the side conditions $\mathcal{C}$, where the meta-conditions in $\mathcal{C}$ are true of $H_{j_1}, \ldots, H_{j_l}$ and H_i. □

In other words, a progressive Hoare proof is a sequence of triples such that every triple in the sequence follows from two preceding triples by applying one of the progressive Hoare rules. We notice that the rules (SKIP) and (ASSIG) have no premises, and therefore may appear first in a progressive Hoare proof.

12.2 Correctness

In order to talk about the correctness of progressive Hoare logics defined in the previous section, we need to establish, at least informally, a link between annotated

```
Prog  ::=  Var := Expr
        |  skip
        |  if Constr then Prog1
                  else Prog2 endif
        |  while Constr do Prog1 endwhile
        |  Prog1 ; Prog2
```

Figure 12.1: Simple Non-Annotated Programming Language

and non-annotated programs. We start by noticing that we can easily turn an annotated program P into a non-annotated program $\mathcal{P}$ by simply removing its annotations. Whenever it is the case that the non-annotated program $\mathcal{P}$ has been obtained by removing the annotations from the annotated program P, we shall write $P \rightsquigarrow \mathcal{P}$. Obviously, the two languages have the same expressive power; we shall not go into detail with proving that if $P \rightsquigarrow \mathcal{P}$, then P and $\mathcal{P}$ have the same trace semantics. Given a non-annotated program $\mathcal{P}$, let P be an annotated program such that $P \rightsquigarrow \mathcal{P}$, and let K be a configuration such that $|K| = P$. By abuse of language, we shall also say that $|K| = P$ whenever it is convenient to do so. We shall also say that a progressive configuration K is the progressive semantics of a non-annotated program $\mathcal{P}$ w.r.t. a set of start environments Σ_0 if there exists an annotated program P such that $P \rightsquigarrow \mathcal{P}$, and whose progression w.r.t. Σ_0 is K.

We continue with a set of propositions that establish the soundness of every progressive Hoare rule. More specifically, we show that if the preconditions of a rule are correct w.r.t. some sets of start environments, and if the side conditions of that rule hold, then the conclusion of the rule is correct w.r.t. some relevant set of start environments.

The next proposition establishes the soundness of the rule (PRECOND).

$$\frac{\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}}{\{\mathcal{F}_1'\}\,\mathcal{P}\,\{\mathcal{F}_2\}}\ \ \mathcal{F}_1' \vdash \mathcal{F}_1 \qquad \text{(PRECOND)}$$

$$\frac{\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}}{\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2'\}}\ \ \mathcal{F}_2 \vdash \mathcal{F}_2' \qquad \text{(POSTCOND)}$$

$$\frac{}{\{\mathcal{F}\}\,\texttt{skip}\,\{\mathcal{F}\}} \qquad \text{(SKIP)}$$

$$\frac{}{\{\mathcal{F}\}\,x := E\,\{\mathit{Assign}(x, e, \mathcal{F})\}} \qquad \text{(ASSIG)}$$

$$\frac{\{\mathcal{F}'\}\,\mathcal{P}_1\,\{\mathcal{F}''\} \quad \{\mathcal{F}''\}\,\mathcal{P}_2\,\{\mathcal{F}'''\}}{\{\mathcal{F}'\}\mathcal{P}_1\,;\,\mathcal{P}_2\{\mathcal{F}'''\}} \qquad \text{(SEQ)}$$

$$\frac{\{\mathit{Filter}(C, \mathcal{F})\}\,\mathcal{P}_c\,\{\mathcal{F}_1\} \quad \{\mathit{Filter}(\neg C, \mathcal{F})\}\,\mathcal{P}_a\,\{\mathcal{F}_2\}}{\{\mathcal{F}\}\,\texttt{if}\,C\,\texttt{then}\,\mathcal{P}_c\,\texttt{else}\,\mathcal{P}_a\,\texttt{endif}\,\{\mathcal{F}_1 \sqcup \mathcal{F}_2\}} \qquad \text{(IF)}$$

$$\frac{\{\mathcal{F}_i'\}\,\mathcal{P}\,\{\mathcal{F}_i''\},\ i = 0, 1, 2, \ldots \qquad \mathcal{F}_2 \equiv \mathit{Filter}(C, \mathit{Seq}(\mathcal{F}_0'', \mathcal{F}_1'', \ldots))}{\begin{array}{l}\{\mathcal{F}_1\}\\ \ \texttt{while}\,C\,\texttt{do}\\ \ \ \mathcal{P}\\ \ \texttt{endwhile}\\ \{\mathit{Filter}(\neg C, \mathcal{F}_1 \sqcup \mathit{Collect}(\mathcal{F}_2))\}\end{array} \quad \mathit{Seq}(\mathcal{F}_1', \mathcal{F}_2', \ldots) \equiv \mathit{Filter}(C, \mathit{Before}(\mathcal{F}_1, \mathcal{F}_2))} \qquad \text{(WHILE)}$$

Figure 12.2: Progressive Hoare Logic

12.4 Proposition Let $\mathcal{F}_1, \mathcal{F}_1', \mathcal{F}_2$ be formulas of a formal system $\mathcal{L}^*$, and let $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ be a progressive Hoare triple that is correct. If $\mathcal{L}^*$ is well-defined, and $\mathcal{F}_1' \vdash \mathcal{F}_1$ holds in $\mathcal{L}^*$, then $\{\mathcal{F}_1'\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is correct as well.

Proof: Let K be a progressive semantics of $\mathcal{P}$ such that $K|_{\mathit{first}(K)} \in \lfloor[\![\mathcal{F}_1']\!]\rfloor$. Since $\mathcal{F}_1' \vdash \mathcal{F}_1$ and $\mathcal{L}^*$ is well-defined, $\lfloor[\![\mathcal{F}_1']\!]\rfloor \subseteq \lfloor[\![\mathcal{F}_1]\!]\rfloor$. It follows that $K|_{\mathit{first}(K)} \in \lfloor[\![\mathcal{F}_1']\!]\rfloor$. Since $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is correct, it follows that $K|_{\mathit{last}(K)} \in \lfloor[\![\mathcal{F}_2]\!]\rfloor$. As a result, $\{\mathcal{F}_1'\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is correct. □

The next proposition establishes the soundness of the rule (POSTCOND).

12.5 Proposition Let $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_2'$ be formulas of a formal system $\mathcal{L}^*$, and let $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ be a progressive Hoare triple that is correct w.r.t. some set of start environments Σ_0. If $\mathcal{L}^*$ is well-defined, and $\mathcal{F}_2 \vdash \mathcal{F}_2'$ holds in $\mathcal{L}^*$, then $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2'\}$ is correct w.r.t. Σ_0 as well.

Proof: Let K be a progressive semantics of $\mathcal{P}$ such that $K|_{\mathit{first}(K)} \in \lfloor[\![\mathcal{F}_1]\!]\rfloor$. Since $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2\}$ is correct, it follows that $K|_{\mathit{last}(K)} \in \lfloor[\![\mathcal{F}_2]\!]\rfloor$. Since $\mathcal{F}_2 \vdash \mathcal{F}_2'$ and $\mathcal{L}^*$ is well-defined, $\lfloor[\![\mathcal{F}_2]\!]\rfloor \subseteq \lfloor[\![\mathcal{F}_2']\!]\rfloor$. It follows that $K|_{\mathit{last}(K)} \in \lfloor[\![\mathcal{F}_2']\!]\rfloor$. As a result, $\{\mathcal{F}_1\}\,\mathcal{P}\,\{\mathcal{F}_2'\}$ is correct. □

The next proposition establishes the soundness of the rule (SKIP).

12.6 Proposition Let $\mathcal{F}$ be a formula of a well-defined formal system $\mathcal{L}^*$. The Hoare triple $\{\mathcal{F}\}\,\texttt{skip}\,\{\mathcal{F}\}$ is correct.

Proof: Given the program $\mathcal{P} = \texttt{skip}$, any progression K of $\mathcal{P}$ has the property $K|_{\mathit{first}(K)} = K|_{\mathit{last}(K)}$. It follows immediately that the proposition holds. □

The next proposition establishes the soundness of the rule (ASSIG).

12.7 Proposition Let Σ_0 be a set of start environments and $\mathcal{F}$ a formula of a formal system $\mathcal{L}^*$, such that $\lambda\langle\rangle\,.\,\Sigma_0 \in \lfloor[\![\mathcal{F}]\!]\rfloor$. If $\mathcal{L}^*$ is well defined, and x and E are a program variable and a program expression, respectively, then the progressive Hoare triple $\{\mathcal{F}\}\, x := E\, \{Assign(x, E, \mathcal{F})\}$ is correct w.r.t. Σ_0.

Proof: Given the program $\mathcal{P} = x := E$, any progression K of $\mathcal{P}$ has the property $K|_{last(K)} = assign(x, E, K|_{first(K)})$. Let K be a progression such that $K|_{first(K)} \in \lfloor[\![\mathcal{F}]\!]\rfloor$. Since $\mathcal{L}^*$ is well-defined, we have that for every formula $\mathcal{F}$, $\widehat{assign}(x, E, [\![\mathcal{F}]\!]) \subseteq [\![Assign(x, E, \mathcal{F})]\!]$, which is equivalent to $\lfloor\widehat{assign}(x, E, [\![\mathcal{F}]\!])\rfloor \subseteq \lfloor[\![Assign(x, E, \mathcal{F})]\!]\rfloor$. Since $assign(x, E, K|_{first(K)}) \in \lfloor\widehat{assign}(x, E, [\![\mathcal{F}]\!])\rfloor$, it follows immediately that the proposition holds. □

The next proposition establishes the soundness of the rule (SEQ).

12.8 Proposition Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two programs, $\mathcal{F}, \mathcal{F}', \mathcal{F}''$ be formulas of a formal system $\mathcal{L}^*$, and assume that the progressive Hoare triples $\{\mathcal{F}\}\,\mathcal{P}_1\,\{\mathcal{F}'\}$ and $\{\mathcal{F}'\}\,\mathcal{P}_2\,\{\mathcal{F}''\}$ are correct. Then, $\{\mathcal{F}\}\,\mathcal{P}_1 \fatsemi \mathcal{P}_2\,\{\mathcal{F}''\}$ is correct.

Proof: Since $\{\mathcal{F}\}\,\mathcal{P}_1\,\{\mathcal{F}'\}$ is correct, for all progressions K_1 of $\mathcal{P}_1$ such that $K_1|_{first(K_1)} \in \lfloor[\![\mathcal{F}]\!]\rfloor$, we also have that $K_1|_{last(K_1)} \in \lfloor[\![\mathcal{F}']\!]\rfloor$. Let K_2 be a progression of $\mathcal{P}_2$, such that $K_2|_{first(K_2)} = K_1|_{last(K_1)}$. Since $\{\mathcal{F}'\}\,\mathcal{P}_2\,\{\mathcal{F}''\}$ is correct, it follows that $K_2|_{last(K_2)} \in \lfloor[\![\mathcal{F}'']\!]\rfloor$. Now, $K_1 \fatsemi K_2$ is a progression of the program $\mathcal{P}_1 \fatsemi \mathcal{P}_2$, and $K_1 \fatsemi K_2|_{first(K_1 \fatsemi K_2)} = K_1|_{last(K_1)}$ and $K_1 \fatsemi K_2|_{last(K_1 \fatsemi K_2)} = K_2|_{last(K_2)}$. It follows that $\{\mathcal{F}\}\,\mathcal{P}_1 \fatsemi \mathcal{P}_2\,\{\mathcal{F}''\}$ is correct. □

The next proposition establishes the soundness of the progressive Hoare rule (IF).

12.9 Proposition Let $\mathcal{P}_c$ and $\mathcal{P}_a$ be two programs, and let $\mathcal{F}, \mathcal{F}_1, \mathcal{F}_2$ be formulas of a well-defined formal system $\mathcal{L}^*$. Given a program constraint C, assume that the progressive Hoare triples $\{\mathit{Filter}(C, \mathcal{F}]\, \mathcal{P}_c\, \{\mathcal{F}_1\}$ and $\{\mathit{Filter}(\neg C, \mathcal{F}]\, \mathcal{P}_a\, \{\mathcal{F}_2\}$ are correct. Then, the progressive Hoare triple $\{\mathcal{F}]\, \texttt{if}\, C\, \texttt{then}\, \mathcal{P}_c\, \texttt{else}\, \mathcal{P}_a\, \texttt{endif}\, \{\mathcal{F}_1 \sqcup \mathcal{F}_2\}$ is correct.

Proof: Let $\Psi \in \lfloor [\![\mathcal{F}]\!] \rfloor$ be an indexed set covered by the interpretation of $\mathcal{F}$. Denote by K_c and K_a the progressive semantics of $\mathcal{P}_c$ and $\mathcal{P}_a$, respectively, such that $K_c|_{\mathit{first}(K_c)} = \mathit{filter}(C, \Psi)$ and $K_a|_{\mathit{first}(K_a)} = \mathit{filter}(\neg C, \Psi)$. Clearly, $\mathit{filter}(C, \Psi) \in \lfloor [\![\mathit{Filter}(C, \mathcal{F})]\!] \rfloor$ and $\mathit{filter}(\neg C, \Psi) \in \lfloor [\![\mathit{Filter}(\neg C, \mathcal{F})]\!] \rfloor$, and since $\{\mathit{Filter}(C, \mathcal{F}]\, \mathcal{P}_c\, \{\mathcal{F}_1\}$ and $\{\mathit{Filter}(\neg C, \mathcal{F}]\, \mathcal{P}_a\, \{\mathcal{F}_2\}$ are correct, it follows that $K_c|_{\mathit{last}(K_c)} \in \lfloor [\![\mathcal{F}_1]\!] \rfloor$ and $K_a|_{\mathit{last}(K_a)} \in \lfloor [\![\mathcal{F}_2]\!] \rfloor$. On the other hand, $\langle l_s, \Psi \rangle\, \texttt{if}\ C\ \texttt{then}\ K_c\ \texttt{else}\ K_a\ \texttt{endif}\, \langle l_f, K_c|_{\mathit{last}(K_c)} \cup K_a|_{\mathit{last}(K_a)} \rangle$, where l_s and l_f are labels that do not appear in K_c and K_a, is a progression of $\texttt{if}\, C\, \texttt{then}\, \mathcal{P}_c\, \texttt{else}\, \mathcal{P}_a\, \texttt{endif}$. $K_c|_{\mathit{last}(K_c)} \cup K_a|_{\mathit{last}(K_a)} \in \lfloor [\![\mathcal{F}_1 \sqcup \mathcal{F}_2]\!] \rfloor$, and since Ψ is an arbitrary element of $\lfloor [\![\mathcal{F}]\!] \rfloor$, it follows that the triple $\{\mathcal{F}]\, \texttt{if}\, C\, \texttt{then}\, \mathcal{P}_c\, \texttt{else}\, \mathcal{P}_a\, \texttt{endif}\, \{\mathcal{F}_1 \sqcup \mathcal{F}_2\}$ is correct. □

The next proposition establishes the soundness of the progressive Hoare rule (WHILE).

12.10 Proposition Let $\mathcal{P}_b$ be a program, and let $\{\mathcal{F}'_i \mid i \geq 0\}$ and $\{\mathcal{F}''_i \mid i \geq 0\}$ be two sets of formulas of a well-defined formal system $\mathcal{L}^*$. Assume that the progressive Hoare triples $\{\mathcal{F}'_i]\, \mathcal{P}_b\, \{\mathcal{F}''_i\}$ are correct, for all $i \geq 0$. Let now C be a program constraint, $\mathcal{F}_1$ and $\mathcal{F}_2$ be two formulas of $\mathcal{L}^*$, and assume the following statements are theorems in $\mathcal{L}^*$:

a) $\mathcal{F}_2 \equiv \mathit{Seq}(\mathcal{F}''_0, \mathcal{F}''_1, \ldots)$

b) $Seq(\mathcal{F}'_0, \mathcal{F}'_1, \ldots) \equiv Filter(C, Before(\mathcal{F}_1, \mathcal{F}_2))$

Then, the progressive Hoare triple $\{\mathcal{F}_1\}\,\texttt{while}\,C\,\texttt{do}\,\mathcal{P}_b\,\texttt{endwhile}\,\{Filter(C, \mathcal{F}_1 \sqcup Collect(\mathcal{F}_2))\}$ is correct.

Proof: Let K be a progression of the program $\texttt{while}\,C\,\texttt{do}\,\mathcal{P}_b\,\texttt{endwhile}$, such that $K|_{first(K)} \in \lfloor[\![\mathcal{F}]\!]\rfloor$. Then, K has the form $\langle l_s, \Psi_s\rangle\,\texttt{while}\,C\,\texttt{do}\,K_b\,\texttt{endwhile}\,\langle l_f, \Psi_f\rangle$, where l_s and l_f are labels, and Ψ_s and Ψ_f are indexed sets. K is a fixpoint of T^p, which entails that $K_b|_{first(K_b)} = filter(C, before(\Psi_s, K_b|_{last(K_b)}))$. We shall now prove by induction that $K_b|_{last(K_b)}(i) \in \lfloor[\![\mathcal{F}''_i]\!]\rfloor$. Since $Seq(\mathcal{F}'_1, \mathcal{F}'_2, \ldots) \equiv Filter(C, Before(\mathcal{F}_1, \mathcal{F}_2))$, it follows that $K_b|_{first(K_b)}(0) \in \lfloor[\![\mathcal{F}'_0]\!]\rfloor$. Since $\{\mathcal{F}'_0\}\,\mathcal{P}_b\,\{\mathcal{F}''_0\}$ is a correct triple, it follows that $K_b|_{last(K_b)}(0) \in \lfloor[\![\mathcal{F}''_0]\!]\rfloor$, which proves the base case. Assume now that $K_b|_{last(K_b)}(i) \in \lfloor[\![\mathcal{F}''_i]\!]\rfloor$, for some $i \geq 0$. From the fact that $Seq(\mathcal{F}'_1, \mathcal{F}'_2, \ldots) \equiv Filter(C, Before(\mathcal{F}_1, \mathcal{F}_2))$, and that K_b is a progression of $\mathcal{P})b$, it follows that $K_b|_{first(K_b)}(i{+}1) \in \lfloor[\![\mathcal{F}'_{i+1}]\!]\rfloor$. Since $\{\mathcal{F}'_{i+1}\}\,\mathcal{P}_b\,\{\mathcal{F}''_{i+1}\}$ is a correct triple, then $K_b|_{last(K_b)}(i+1) \in \lfloor[\![\mathcal{F}''_{i+1}]\!]\rfloor$, which proves the induction case. As a result, $\Psi_f = filter(C, collect(K_b|_{last(K_b)})) \in \lfloor[\![Filter(C, Collect(\mathcal{F}_2))]\!]\rfloor$. It follows that $\{\mathcal{F}_1\}\,\texttt{while}\,C\,\texttt{do}\,\mathcal{P}_b\,\texttt{endwhile}\,\{Filter(C, \mathcal{F}_1 \sqcup Collect(\mathcal{F}_2))\}$ is a correct triple. □

The following theorem is the highlight of this section, showing that the definition of a progressive Hoare proof is sound.

12.11 Theorem Let $\mathcal{P}$ be a program and $\mathcal{F}$, $\mathcal{F}'$ two formulas of a well-defined formal system $\mathcal{L}^*$. If there exists a progressive Hoare proof whose last triple is $H = \{\mathcal{F}\}\,\mathcal{P}\,\{\mathcal{F}'\}$, then H is correct w.r.t. all sets of start environments Σ_0 such that $\lambda\langle\rangle\,.\,\Sigma_0 \in \lfloor[\![\mathcal{F}]\!]\rfloor$.

Proof: We prove this proposition by induction on the length of the progressive Hoare proof. For a proof of length 1, the formula in the proof must be an instance of either of the progressive Hoare rules (SKIP) and (ASSIG), defined in Figure 12.2. Propositions 12.6 and 12.7 show that such formulas are correct. If the length of the progressive Hoare proof is longer than 1, then, by the induction hypothesis, all the formulas in the proof, except possibly the last one, are correct. Then the last formula in the proof must be either an instance of the (SKIP) and (ASSIG) rules, in which case Propositions 12.6 and 12.7 apply, or it is obtained by the application of one of the rules (PRECOND), (POSTCOND), (SEQ), (IF), or (WHILE), in which case Propositions 12.4, 12.5, 12.8, 12.9, 12.10 apply. It follows that the last formula in the proof is correct, which proves the induction case and the proposition. □

In the following chapters, we shall introduce two family description languages and explore the progressive Hoare calculi that are based on them.

Chapter 13

A Liveness-Aware Description Language

In this chapter we introduce a family description language that is "liveness aware", in the sense that its interpretation is able to distinguish only between families that contain the empty set and families that do not. As a result, this language is strong enough to prove total correctness. The language uses first order formulas, employed in a manner very similar to classic Hoare logic. Two important differences exist though. On one hand, index variables play an active role, expressing an abstraction of the sequencing of environments occurring at every program point, and the use of the "*" annotation, which indicates the fact that we have exact knowledge of whether a program point will be reached or not. For example, consider the starred formula $\mathcal{F} = \lambda\langle\nu\rangle\,.\,(\nu\%2 = 0 \wedge x > 0)^*$ that may be attached to a program point l, inside an `if` statement, that is in turn nested inside a `while` loop. The formula $\mathcal{F}$ states that during the even repetitions of the body of the `while` loop, program point l will definitely be reached, and the variable x will be positive. However, during the odd repetition of the `while` loop body, program point l is definitely

$$
\begin{array}{rcll}
CT & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \cdots & \text{(constants)} \\
PV & ::= & x \mid y \mid z \mid \cdots & \text{(program variables)} \\
IV & ::= & \nu_0 \mid \nu_1 \mid \nu_2 \mid \cdots & \text{(index variables)} \\
AV & ::= & \delta_0 \mid \delta_1 \mid \cdots \mid \cdots & \text{(auxiliary variables)} \\
FS & ::= & + \mid - \mid * \mid / \mid f \mid g \mid \cdots & \text{(function symbols)} \\
PS & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid p \mid q \mid \cdots & \text{(predicate symbols)} \\
T & ::= & CT \mid PV \mid IV \mid AV \mid FS(T_1, \ldots, T_n) & \text{(terms)} \\
SF & ::= & PS(T_1, \ldots, T_n) \mid SF_1 \wedge SF_2 \mid SF_1 \vee SF_2 \mid \neg SF' \mid & \\
 & & SF_1 \rightarrow SF_2 \mid SF_1 \leftrightarrow SF_2 \mid \forall\, AV\,.\,SF' \mid \exists\, AV\,.\,SF' & \text{(simple formulas)} \\
LAL & ::= & \lambda\langle IV_1, \ldots, IV_n\rangle\,.\,SF \mid & \text{(only program variables may appear free in } LAL\text{)} \\
 & & \lambda\langle IV_1, \ldots, IV_n\rangle\,.\,SF^* \mid &
\end{array}
$$

Figure 13.1: Liveness Aware Family Description Language

not reached (possibly because the other branch of the `if` statement is taken.) In contrast, consider the non-starred formula $\mathcal{F}' = \lambda\langle\nu\rangle\,.\,\nu\%2 = 0 \wedge x > 0$, attached to the same program point l. This formula states that program point l *may* be reached during the even repetitions of the `while` loop body and, if that happens, the value of x will be positive. Clearly, $\mathcal{F}'$ is less precise than $\mathcal{F}$. In order to facilitate the definition of the progressive Hoare rules, the language also has operators similar to the family progressive operators defined in Figure 11.4. In the formal system that we define on top of the family description language, our main concern is to preserve the "*" annotation through these operators.

13.1 Language

The liveness-aware family description language LAL is defined in Figure 13.1. The terms of this language are defined by the non-terminal T, and are either constants, variables, or functions applied to other terms. In this language we distinguish between *program variables* $x, y, z, \ldots$, which are variables appearing in the program at hand, *index variables* $\nu_0, \nu_1, \nu_2, \ldots$, which model the indices used by families to represent the sequencing of properties, and *auxiliary variables* $\delta_0, \delta_1, \delta_2, \ldots$, which are introduced for the sole purpose of increasing the expressive power of the language. Using predicate symbols defined by the non-terminal *PS*, and terms, we can define simple formulas, represented by the non-terminal *SF*. Such formulas resemble first order logic formulas, with the only difference that we distinguish between types of variables. The formulas defined by *SF*, where all the auxiliary variables appear quantified, can be used in building formulas of LAL. Since φ resembles a first order formula, we shall take advantage of this fact when defining the interpretation of LAL formulas. To this purpose, we shall denote by $\forall\varphi$ the first order closure of φ in which all variables appear quantified, and we shall denote by $\models \forall\varphi$ the fact that $\forall\varphi$ is a theorem in first order logic. Also, it is convenient to define $\forall_\nu\varphi$ the closure of φ in which the index variables appear quantified, but in which program variables are still free, and by $\sigma \models \forall_\nu\varphi$ the fact that the first order formula $\forall_\nu\varphi$ is true w.r.t. the environment σ, which acts as a valuation for the program variables that appear free in $\forall_\nu\varphi$. Similarly, it is convenient to define $\forall_x\varphi$ the closure of φ in which program variables appear quantified, but in which index variables are still free. We also notice that program expressions and program constraints, as defined by the non-terminals **Expr** and **Constr** in Figure 4.1, are terms and simple formulas of LAL, respectively.

A LAL formula is typically either an expression of the form $\lambda\langle\nu_1\nu_2\cdots\nu_k\rangle\,.\,\varphi$ or an expression of the form $\lambda\langle\nu_1\nu_2\cdots\nu_k\rangle\,.\,\varphi^*$, where φ is a simple formula in which all auxiliary variables appear quantified. The number of index variables k is called the *arity* of the formula. Given a formula $\mathcal{F}$, we denote its arity by $arity(\mathcal{F})$. The first type of formulas, without a star superscript, shall be called *liveness-insensitive formulas*, while the second type, which has a star superscript, shall be called *liveness-sensitive fromulas*. Intuitively, in the family that represents the interpretation of a liveness-sensitive formula, each slice either contains only the empty set, or does not contain the empty set at all. For the liveness-insensitive formulas, each slice of the interpretation is subset-closed, and includes the empty set.

Finally, we need to define the progressive meta-operators used in the definition of the progressive Hoare rules. These meta-operators appear in meta-formulas that can be translated into formulas of LAL. When defining such operators, we have to bear in mind that a LAL formula is either a liveness-sensitive formula, or a liveness-insensitive formula. Whenever it is convenient, we shall provide case-based definitions for our operators. Some of the operators, namely *Before* and $\sqcup$ are only defined for certain combinations of formulas. For example, the $\sqcup$ operator can only be used between formulas with the same arity. In what follows, we shall consider only formulas where this operators can be applied. This assumption is also extended to symbolic configurations in the next chapter.

We start with the *Assign* operator, defined by the following three expressions, in which $k \geq 0$, x is a program variable, and E is a program expression, φ is a simple expression, and δ is an auxiliary variable that does not appear in φ.

$$\begin{aligned} \mathit{Assign}(x, E, \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi) &= \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \exists\delta \,.\, (\varphi[\delta/x] \wedge x = (E[\delta/x])) \\ \mathit{Assign}(x, E, \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*) &= \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, (\exists\delta \,.\, (\varphi[\delta/x] \wedge x = (E[\delta/x])))^* \end{aligned}$$

We continue with the definition of *Filter*, defined by the following three expressions. The symbol φ retains its role of representing a simple expression, C is a program constraint, and $k \geq 0$.

$$\begin{aligned} \mathit{Filter}(C, \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi) &= \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, (\varphi \wedge C) \\ \mathit{Filter}(C, \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*) &= \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, (\varphi \wedge C)^*, \quad \text{if } \models (\forall(C \rightarrow \varphi)) \vee (\forall(C \rightarrow \neg\varphi)) \\ \mathit{Filter}(C, \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*) &= \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, (\varphi \wedge C), \quad \text{if } \models (\neg\forall(C \rightarrow \varphi)) \vee (\forall(C \rightarrow \neg\varphi)) \end{aligned}$$

Next, we define the *Before* meta-operator in a case-based manner. In this definition, φ_1 and φ_2 are simple formulas, and $k \geq 0$ is an integer. The $\mathit{Before}(\mathcal{F}_1, \mathcal{F}_2)$ expression is defined only for formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ such that $\mathit{arity}(\mathcal{F}_1) = \mathit{arity}(\mathcal{F}_2) - 1$.

$$\begin{aligned} &\mathit{Before}(\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1, \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2) = \\ &\qquad \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, ((\nu_{k+1} = 0 \rightarrow \varphi_1) \wedge (\nu_{k+1} > 0 \rightarrow \varphi_2[(\nu_{k+1} - 1)/\nu_{k+1}])) \\ &\mathit{Before}(\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1^*, \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2^*) = \\ &\qquad \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, ((\nu_{k+1} = 0 \rightarrow \varphi_1) \wedge (\nu_{k+1} > 0 \rightarrow \varphi_2[(\nu_{k+1} - 1)/\nu_{k+1}]))^* \\ &\mathit{Before}(\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1, \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2^*) = \\ &\qquad \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, (\nu_{k+1} = 0 \rightarrow \varphi_1 \wedge \nu_{k+1} > 0 \rightarrow \varphi_2[(\nu_{k+1} - 1)/\nu_{k+1}]) \\ &\mathit{Before}(\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1^*, \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2) = \end{aligned}$$

We note that, in order to be useful, the *Before* operator must have as arguments two formulas such that the second one has one extra index variable argument. We continue with the definition of *Collect*, given in a case-based manner. The symbols k, $\nu_1, \ldots, \nu_k$, φ and δ have the same meaning as above. The operator *Collect* is defined only for formulas of arity strictly greater than 0.

$$\begin{aligned} \mathit{Collect}(\lambda\langle \nu_1 \cdots \nu_k \nu_{k+1}\rangle \,.\, \varphi) &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \exists\delta \,.\, \varphi[\delta/\nu_{k+1}] \\ \mathit{Collect}(\lambda\langle \nu_1 \cdots \nu_k \nu_{k+1}\rangle \,.\, \varphi^*) &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, (\exists\delta \,.\, \varphi[\delta/\nu_{k+1}])^* \end{aligned}$$

The last expression is useful to define the *Collect* operator for LAL formulas of the form $\lambda\langle\rangle \,.\, \varphi$ or $\lambda\langle\rangle \,.\, \varphi^*$. We continue with the definition of $\sqcup$, which requires operands of the same arity.

$$\begin{aligned} \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \;\sqcup\; \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_2 &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \vee \varphi_2 \\ \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1^* \;\sqcup\; \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_2^* &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, (\varphi_1 \vee \varphi_2)^* \\ \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1^* \;\sqcup\; \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_2 &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \vee \varphi_2 \\ \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \;\sqcup\; \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_2^* &= \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \vee \varphi_2 \end{aligned}$$

Finally, we provide the definition of the *Seq* operator, which is defined only for formulas of the same arity.

$$\begin{aligned} \mathit{Seq}(\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_m, \ldots) &= \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1}\rangle \,.\, \varphi, \quad \text{if there exists } \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi \in \text{LAL s.t.} \\ &\qquad \mathcal{F}_i = \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi[\nu_{k+1}/i],\ i \geq 0 \\ \mathit{Seq}(\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_m, \ldots) &= \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1}\rangle \,.\, \varphi^*, \quad \text{if there exists } \lambda\langle \nu_1 \cdots \nu_k\rangle \,.\, \varphi^* \in \text{LAL s.t.} \end{aligned}$$

We continue with providing an interpretation for the liveness aware language.

13.2 Interpretation

LAL formulas are interpreted as indexed families. Given a liveness insensitive formula $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi$, where φ is a simple formula in which index variables $\nu_1, \ldots, \nu_k$ appear free, no auxiliary variables appear free, and $k \geq 0$, its interpretation is defined as:

$$[\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] = \lambda\langle\mu_1 \cdots \mu_k\rangle \,.\, \{\Sigma \,|\, \text{forall } \sigma \in \Sigma, \sigma \models (\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k])\}.$$

In the definition above, the notation $\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]$ stands for the formula obtained from φ by replacing the index variable ν_i by the natural number μ_i, for all i, $1 \leq i \leq k$. We call the expression $[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]$ a *substitution.*

Given a liveness sensitive formula $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*$, where φ is a simple formula in which index variables $\nu_1, \ldots, \nu_k$ appear free, and no auxiliary variables appear free, and $k \geq 0$ its interpretation is defined as:

$$[\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*]\!] = \lambda\langle\mu_1 \cdots \mu_k\rangle \,.\, \begin{cases} \{\emptyset\}, & \text{if } \models \forall(\neg\varphi[\mu_1/\nu_1, \ldots \mu_k/\nu_k]) \\ \{\Sigma \,|\, \Sigma \neq \emptyset \text{ and forall } \sigma \in \Sigma, \sigma \models (\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k])\}, & \\ & \text{otherwise} \end{cases}$$

Next, we define a formal system on top of the liveness aware language. The formal system will help establishing the validity of formulas of the form $\mathcal{F}_1 \vdash \mathcal{F}_2$, which is necessary in order to define a concrete progressive Hoare logic.

13.3 Formal System

We now define a formal system called LAL*, on top of the liveness aware language. This formal system deals with formulas of the type $\mathcal{F}_1 \vdash \mathcal{F}_2$, where $\mathcal{F}_1, \mathcal{F}_2 \in$ LAL, and $\vdash$ is a new symbol. The purpose of this formal system is to be able to prove

(LAL1) $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^* \vdash \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi$

(LAL2) $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1 \vdash \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_2$
$\text{if } \models \forall(\varphi_1 \rightarrow \varphi_2)$

(LAL3) $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1^* \vdash \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_2^*$
$\text{if } \models \forall(\varphi_1 \rightarrow \varphi_2) \text{ and } \models \forall((\forall_x \neg\varphi_1) \rightarrow (\forall_x \neg\varphi_2))$

Figure 13.2: LAL* Axioms

the $\mathcal{F}_1 \vdash \mathcal{F}_2$ formulas whenever $[\![\mathcal{F}_1]\!] \subseteq [\![\mathcal{F}_2]\!]$. The formal system consists of the language LAL, an inference rule, and a set of axioms. The inference rule has the form

$$\frac{\mathcal{F}_1 \vdash \mathcal{F}_2 \quad \mathcal{F}_2 \vdash \mathcal{F}_3}{\mathcal{F}_1 \vdash \mathcal{F}_3},$$

being similar to the modus ponens rule in classic first order logic.

The set of axioms are given in Figure 13.2. They have the general form

$$\mathcal{F}_1 \vdash \mathcal{F}_2 \quad \text{if } \mathcal{C}$$

where $\mathcal{F}_1, \mathcal{F}_2$ are two LAL formulas. Typically, inside these two formulas there are embedded several first order formulas $\varphi_1, \varphi_2, \ldots, \varphi_p$. The meta-condition $\mathcal{C}$ is a first order theorem about $\varphi_1, \varphi_2, \ldots, \varphi_p$ that needs to hold in first order logic, in order for $\mathcal{F}_1 \vdash \mathcal{F}_2$ to be a valid axiom. The axioms are mainly concerned with manipulating explicit formulas, or defining relationships between non-explicit formulas and explicit ones. Axiom LAL1 formalizes the general intuition that starred formulas are more precise than non-starred formulas. Indeed, starred formulas, whenever attached to a program point, besides a conservative approximation of the set of environments occurring at that point, express the knowledge of whether that program point is reached or not. Axioms LAL2 and LAL3 show that the relationships

between explicit LAL formulas rely on the relationships between the first order formulas embedded within them. The LAL3 axiom deserves special attention, since it is one case in which we have to be careful about preserving the star annotation. As defined in the previous section, the interpretation of a starred formula $\mathcal{F}$ is a indexed family F with the property that for all $\tilde{\mu} \in \mathbf{Idx}$, $F(\tilde{\mu})$ either contains only the empty set, or does not contain the empty set at all. Consider two starred formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, and assume that $\mathcal{F}_1 \vdash \mathcal{F}_2$ holds. Denote by F_1 and F_2 the interpretations of $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. It must be the case that $F_1(\tilde{\mu}) \subseteq F_2(\tilde{\mu})$ for all $\tilde{\mu} \in \mathbf{Idx}$. However, since $\mathcal{F}_1$ and $\mathcal{F}_2$ are starred formulas, the families F_1 and F_2 must also have the following property: $F_2(\tilde{\mu}) = \{\emptyset\}$ whenever $F_1(\tilde{\mu}) = \{\emptyset\}$. This is tantamount to the first order formula $\forall((\forall_x \neg\varphi_1) \rightarrow (\forall_x \neg\varphi_2))$ being a theorem of first order logic.

Next, we define the concepts of proof and theorem for LAL*, in the standard way. A *proof* in LAL* is a sequence of formulas of the form $\mathcal{F}_1 \vdash \mathcal{F}'_1, \mathcal{F}_2 \vdash \mathcal{F}'_2, \ldots, \mathcal{F}_k \vdash \mathcal{F}'_k$, such that every $\mathcal{F}_i \vdash \mathcal{F}'_i$, $1 \leq i \leq k$ is either an instance of an axiom, or it follows from two previous formulas via the inference rule. The last formula of the sequence representing the proof is a *theorem* of LAL*.

At this point, we can prove that our formal system is well-defined. Since the well-definedness definition specifies seven conditions, we shall prove that each of these conditions holds in a separate proposition, and then, use these partial results to prove the overall statement that the LAL* formal system is well defined. We start by giving two remarks that shall be useful in the proofs.

13.1 Remark Let $\mathcal{F}$ be a LAL formula that is either liveness-insensitive, i.e. of the form $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi$, or liveness-sensitive, i.e. of the form $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*$. Denote by Ψ an indexed set such that $\Psi \in \lfloor [\![\mathcal{F}]\!] \rfloor$. The following two conditions hold:

a) $\Psi(\tilde{\mu}) = \emptyset$, if $size(\tilde{\mu}) \neq k$.

b) For all $\sigma \in \Psi(\mu_1 \cdots \mu_k)$, $\sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]$.

If $\mathcal{F}$ is a liveness-sensitive formula, the following condition holds on top of the two conditions above.

c) If for a given $\tilde{\mu}$, there exists an environment σ such that $\sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]$, then $\Psi(\tilde{\mu}) \neq \emptyset$.

□

13.2 Remark Let ξ be a property of indexed sets such that $\{\Psi \,|\, \xi(\Psi)\}$ is a well-defined set. Denote by Φ the indexed family $\lceil\{\Psi \,|\, \xi(\Psi)\}\rceil$. Then, $\Phi(\tilde{\mu}) = \{\Psi(\tilde{\mu}) \,|\, \xi(\Psi)\}$, for all $\tilde{\mu} \in \mathbf{Idx}$. Moreover, let η be a mapping from indexed sets, such that $\{\eta(\Psi) \,|\, \xi(\Psi)\}$ is a well-defined set, and denote by Φ the indexed set $\lceil\{\eta(\Psi) \,|\, \xi(\Psi)\}\rceil$. Then $\Phi(\tilde{\mu}) = \{(\eta(\Psi))(\tilde{\mu}) \,|\, \xi(\Psi)\}$, for all $\tilde{\mu} \in \mathbf{Idx}$. □

We continue with proving condition (a) of Definition 12.1.

13.3 Proposition For all program variables x, program expressions E, and formulas $\mathcal{F} \in$ LAL, we have $\widehat{assign}(x, E, [\![\mathcal{F}]\!]) \subseteq [\![Assign(x, E, \mathcal{F})]\!]$.

Proof: Relegated to Appendix A, on page 315.

We continue with proving condition (b) of Definition 12.1.

13.4 Proposition For all program constraints C, and formulas $\mathcal{F} \in$ LAL, we have $\widehat{filter}(C, [\![\mathcal{F}]\!]) \subseteq [\![Filter(C, \mathcal{F})]\!]$.

Proof: Relegated to Appendix A, on page 317.

Next, we prove condition (c) of Definition 12.1.

13.5 Proposition For all formulas $\mathcal{F}_1, \mathcal{F}_2 \in$ LAL, we have $\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]) \subseteq [\![Before(\mathcal{F}_1, \mathcal{F}_2)]\!]$.

Proof: Relegated to Appendix A, on page 321.

Now we prove condition (d) of Definition 12.1.

13.6 Proposition For all formulas $\mathcal{F} \in$ LAL, we have $\widehat{collect}([\![\mathcal{F}]\!]) \subseteq [\![Collect(\mathcal{F})]\!]$.

Proof: Relegated to Appendix A, on page 324.

We continue with proving condition (e) of Definition 12.1.

13.7 Proposition For all formulas $\mathcal{F}_1, \mathcal{F}_2 \in$ LAL, we have $[\![\mathcal{F}_1]\!] \mathbin{\widehat{\cup}} [\![\mathcal{F}_2]\!] \subseteq [\![F_1 \sqcup F_2]\!]$.

Proof: Relegated to Appendix A, on page 326.

Next we prove condition (f) of Definition 12.1.

13.8 Proposition For all formulas $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots \in$ LAL, we have $seq([\![\mathcal{F}_0]\!], [\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!], \ldots) \subseteq [\![Seq(\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots)]\!]$.

Proof: Relegated to Appendix A, on page 328.

We now prove condition (g) of Definition 12.1.

13.9 Proposition For all formulas $\mathcal{F}, \mathcal{F}' \in$ LAL, we have $\mathcal{F} \vdash \mathcal{F}'$ is a theorem in $\mathcal{L}^*$ only if $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$.

Proof: Relegated to Appendix A, on page 329.

Finally, we are able to assemble the proofs in the previous seven propositions into a result that states that the formal system LAL* is well-defined.

```
while x > 1 do
    if x % 2 = 0 then
        x := x/2
    then
        x := x + 1
    endif
endwhile
```

Figure 13.3: Program Subjected to Termination Proof

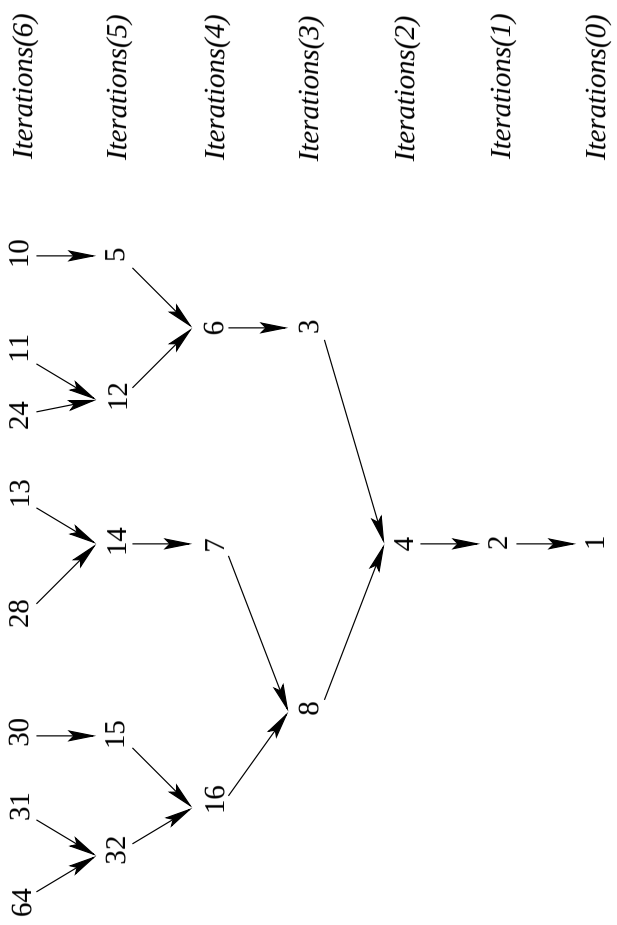
13.10 Theorem The LAL* formal system is well-defined.

Proof: Propositions 13.3 – 13.9 prove all the conditions in Definition 12.1 for the formal system LAL*. As a result, LAL* is well-defined. □

13.4 Liveness Aware Language Example

It is now the time to put the LAL-based progressive Hoare calculus to work. We shall consider an example program whose termination is far from obvious, and show how to construct a progressive Hoare proof from which it can be inferred not only that the program terminates, but also the sequence of values that occur at every program point for the variable used in the program.

Consider the program in Figure 13.3. The program executes a loop for as long as variable x is greater than 1. Inside the loop, the variable is halved if it has an even value, and it is incremented by 1 if it has an odd value. We shall assume that the initial value of x is positive, but unknown, and this fact shall be reflected in the precondition we infer for our program. Since the value of x is not monotonically



$$\mathit{Iterations}(\nu_0) = \{x_0 \,|\, \lceil \log_2 x_0 \rceil - 1 + \mathit{ones}(2^{\lceil \log_2 x_0 \rceil + 1} - x_0)\}$$

Figure 13.4: Termination of Program in Figure 13.3

decreasing, the termination of this program is not obvious. We shall first argue informally that this program terminates, and then we shall construct a progressive Hoare proof from which the termination of the program (and not only) can be inferred. In order to prove termination, we resort to proving the following statement:

> Denote by $ones(z)$ the number of bits set to 1 in the binary representation of the integer z. Assuming that the program starts with an initial value x_0 for the program variable x, the program terminates after $\lceil \log_2 x_0 \rceil - 1 + ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0)$.

The proof is based on the following easy to verify algebraic equalities:

$$\text{if } x_0 \text{ is odd:} \quad \lceil \log_2 x_0 \rceil = \lceil \log_2 (x_0 + 1) \rceil \text{ and}$$
$$ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0) = ones(2^{\lceil \log_2 (x_0+1) \rceil + 1} - (x_0 + 1)) - 1$$

$$\text{if } x_0 \text{ is even:} \quad \lceil \log_2 x_0 \rceil = \lceil \log_2 (x_0/2) - 1 \rceil \text{ and}$$
$$ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0) = ones(2^{\lceil \log_2 (x_0/2) \rceil + 1} - (x_0/2)) - 1$$

Let us denote by $\nu_0 = \lceil \log_2 x_0 \rceil - 1 + ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0)$ the number of iterations through the loop for a given x_0.[1] We prove the above claim by induction on ν_0. Assume ν_0 is 0. This entails that x_0 must be 0. When x_0 is 0, there will be 0 iterations through the `while` loop body, which satisfies the claim. Assume now that the claim holds for all $\nu_0 \geq 0$ and consider the case for $\nu_0 + 1$. Let x_0 be an initial value of x such that the program terminates after $\nu_0 + 1$ iterations through the loop. This means that $\nu_0 + 1 = \lceil \log_2 x_0 \rceil - 1 + ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0)$ We have two cases: x_0 may be odd or even. In the case when x_0 is odd, the value of x after one iteration through the loop is $x_0 + 1$. On the other hand, when x_0 is even, the value

[1] The choice of the symbol ν_0 for the number of iterations may seem strange at this point. However, since ν_0 is typically used as a index variable in LAL formulas, the notation will come in handy when we give the progressive Hoare proof of the program.

of x after one iteration through the loop is $x_0/2$. Let us denote by x_1 the value of variable x after the first iteration through the loop. Using the algebraic equalities given above, we have that $\lceil \log_2 x_1 \rceil - 1 + ones(2^{\lceil \log_2 x_1 \rceil + 1} - x_1) = \nu_0$, which proves the induction case. Having understood the principle behind the termination of the program, let us now build a progressive Hoare proof from which termination can be inferred formally. For convenience, we denote by $B(\nu, x)$ the simple formula $\lceil \log_2 x_0 \rceil - 1 + ones(2^{\lceil \log_2 x_0 \rceil + 1} - x_0)$. Using the progressive Hoare rule (ASSIG), we produce the following two correct progressive Hoare triples.

$$\begin{array}{l} \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1,x)\wedge x>1\wedge x\,\%\,2=0)^*\} \\ \quad x := x/2 \qquad\qquad (\text{PH1}) \\ \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1-1,x))^*\} \end{array}$$

$$\begin{array}{l} \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1,x)\wedge x>1\wedge a\,\%\,2\neq 0)^*\} \\ \quad x := x+1 \qquad\qquad (\text{PH2}) \\ \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1-1,x))^*\} \end{array}$$

Using the triples (PH1) and (PH2), and the (IF) progressive Hoare rule, we produce the following progressive Hoare triple.

$$\begin{array}{l} \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1,x)\wedge x>1)^*\} \\ \quad \texttt{if } x\,\%\,2=0 \texttt{ then} \\ \quad\quad x := x/2 \\ \quad \texttt{else} \qquad\qquad (\text{PH3}) \\ \quad\quad x := x+1 \\ \quad \texttt{endif} \\ \{\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0-\nu_1-1,x))^*\} \end{array}$$

Next, we shall produce the Hoare triple corresponding to the entire program, by using the (WHILE) progressive rule. We prepare the ground for applying the (WHILE)

rule by defining the following notations.

$$\mathcal{F}'_i = \lambda\langle\nu_0\rangle\,.\,(B(\nu_0 - i, x) \wedge x > 1)^*$$

$$\mathcal{F}''_i = \lambda\langle\nu_0\rangle\,.\,(B(\nu_0 - i - 1, x) \wedge x > 1)^*$$

$$\mathcal{F}_2 = \mathit{Seq}(\mathcal{F}'_0, \mathcal{F}'_1, \mathcal{F}'_2, \ldots)$$

We now note that

$$\mathit{Filter}(x > 1, \mathit{Seq}(\mathcal{F}'_0, \mathcal{F}'_1, \mathcal{F}'_2, \ldots)) = \mathit{Filter}(x > 1, \mathit{Before}(\lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*, \mathcal{F}_2)).$$

As a result, we can use the (PH3) Hoare triple and apply the (WHILE) progressive Hoare rule in order to obtain the following triple.

$$\{\lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*\}$$

```
while x > 1 do
    if x % 2 = 0 then
        x := x/2
    else
        x := x + 1
    endif
endwhile
```

$$\{\lambda\langle\nu_0\rangle\,.\,(x = 1)^*\}$$

From this Hoare triple we can infer that the program terminates and variable x has value 1 at the end of the execution. One important thing to notice is the LAL formula that acts as a precondition to this triple. It specifies that the first point of the program is definitely reached, and that the value of x is positive. This information is specified in a stratified way, each value of ν_0 specifying values of x for which the program terminates after executing ν_0 iterations through the loop. An alternative way of specifying this information would have been the LAL

```
while x < 100 do
    while a[x] ≠ 1 do
        x := x + 1
    endwhile
    y := x * x  /* output the value of x here */
    while y ≤ 100 do
        a := a[y ↦ 0]
        y := y + x
    endwhile
    x := x + 1
endwhile
```

Figure 13.5: Primes Program

formula $\lambda\langle\rangle\,.\,x > 0$. However, using this precondition, we wouldn't have achieved a termination proof.

13.5 A Prime Number Program Example

In this section we present a more elaborate example which builds a progressive Hoare proof for a program computing all the primes between 2 and 100. We shall show, in the same proof, that the program computes all the primes *in increasing order*, and that it terminates. This will show that the LAL language can be used not only for proving liveness or termination, but also safety and sequence-based properties. The program is presented in Figure 13.5, and it uses a version of Eratosthenes' sieve algorithm.

The program starts with variable x initialized to 2, and all the elements of the array a initialized to 1. This initializations are not shown in the program, but will be reflected in the final Hoare triple that constitutes the proof of the entire

program. The array a represents the sieve; all its elements of non-prime rank shall be turned to 0. Between the two inner loops, variable x is regarded as the output of the program, we will show that the sequence of values assigned to this variable at either of the program points between the two inner loops is the sequence of all prime numbers between 2 and 100. Intuitively, the program works in the following way. At the beginning of the outer `while` loop, the least rank δ such that $a[\delta] = 1$ is a prime number. For this reason, the first inner `while` loop will search for the next element equal to 1 in the array. Thus, upon exit from the inner `while` loop, the value of x is the "next" prime number. The second `while` loop will strike out the multiples of x (i.e. set $a[\delta]$ to 0, where δ sweeps over the multiples of x), starting with x^2. The process continues until x hits the first prime greater than 100. One deviation from the classic sieve algorithm is that we start striking out the multiples of x starting with x^2. The reason for this is that all multiples of x smaller than x^2 are also multiples of prime numbers smaller than the value of x, and have already been stricken out in the previous iterations through the outer loop.

To make the progressive Hoare triple easier to understand, it is convenient to define the following notations.

$$\begin{aligned} \mathit{prime}(\delta) &\equiv \forall \delta' \,.\, 2 \leq \delta' < \delta \rightarrow \delta \,\%\, \delta' \neq 0 \\ \mathit{prime\_rank}(\delta_1, \delta_2) &\equiv \exists \eta \,.\, (\forall \delta_1 \,.\, \forall \delta_2 \,.\, \delta_1 < \delta_2 \rightarrow \eta(\delta_1) < \eta(\delta_2)) \\ &\quad \wedge\ (\forall \delta \,.\, \mathit{prime}(\eta(\delta)) \\ &\quad \wedge\ (\forall \delta \,.\, \mathit{prime}(\delta) \rightarrow \exists \delta' \,.\, \eta(\delta') = \delta) \\ &\quad \wedge\ \delta_2 = \eta(\delta_1) \end{aligned}$$

The LAL language does not allow for predicate definitions. For this reason, we shall regard the two notations above as meta-predicates: whenever they are used in a formula, we shall assume that the predicate definition is "in-lined" as a replacement for the predicate expression, after the appropriate substitution has been applied.

The predicate $prime(\delta)$ is true whenever its parameter is a prime number. Its definition is very straightforward: δ is not a multiple of any number between 2 and $\delta - 1$. Another convenient notation is the $prime\_rank(\delta_1, \delta_2)$ predicate, which is true if δ_2 is the prime number having rank δ_1 in the increasing sequence of prime numbers. To define this predicate, we need an extension to the LAL language, that allows quantifiable functional variables. In the definition of *prime_rank*, the symbol η is such a functional variable. The definition of *prime_rank* states that there exists a function η that is strictly increasing (first line), and whose numbers are primes (second line), and moreover, all primes are in the range of η (third line), and finally, that δ_2 is $\eta(\delta_1)$ (fourth line).

We shall start our proof by putting a progressive Hoare triple around the first $x := x + 1$ statement. Since this statement is nested inside two while loops, we shall use formulas with two parameters, ν_1 and ν_2, one for each level of nesting. In order to build these formulas, let us understand the configuration of the program just before the statement is executed. At that point, the value of x is somewhere between two primes. All the elements of array a and whose rank is smaller than x and whose value is 1 are primes. Moreover, all the elements in the array whose rank is a multiple of a prime smaller than x have the value 0. The smallest element of a whose value is 1 and whose rank is greater than x is also a prime, and is in fact the next prime that will be output. Since the outer loop outputs one prime number per iteration, the rank of the next prime is ν_1. Also, since the value of x at the bottom of the outer `while` loop, in the previous iteration, was one greater than the prime of rank $\nu_1 - 1$, the current value of x must be the prime of rank $\nu_1 - 1$ plus $\nu_2 + 1$. All this information is encoded in the following progressive Hoare triple, which is an instance of the (ASSIG) rule.

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad a[x] \neq 1 \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge x = \delta_0 + \nu_2 + 1 \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2) \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0))^*\}$$

PRIME-1

$$x := x + 1$$

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad a[x-1] \neq 1 \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge x = \delta_0 + \nu_2 + 2 \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ \exists\delta_0.(prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0))^*\}$$

We now note that the following formula is a theorem in Higher Order Logic.

$$\forall\nu_1\,.\quad (a[x] = 1 \wedge (\exists\nu_2\,.\,(\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge x = \delta_0 + \nu_2 + 2 \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ \exists\delta_0.(prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0))\ \leftrightarrow$$
$$(prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0))$$

Using the triple PRIME-1 as a premise, we can apply the (WHILE) rule, to obtain the following triple.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1-1,x-1)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\leq\delta\leq100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\leq\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2\leq\delta_1<\delta_0^2\wedge\delta_1\delta_2\geq\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0))^*\}$$

`while` $a[x]\neq1$ `do`

$x:=x+1$ PRIME-2

`endwhile`

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1,x)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\leq\delta\leq100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\leq\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2\leq\delta_1<\delta_0^2\wedge\delta_1\delta_2\geq\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0))^*\}$$

The PRIME-2 triple has the following meaning. Whenever the beginning of the program fragment is reached, the value of x is one greater than the last prime that was computed, which is the prime of rank ν_1-1. All the array elements whose ranks are non-primes smaller than x are set to 0. Also, all the array elements whose ranks are multiples of prime numbers smaller than x are set to 0. And obviously, since this is the first program point inside the outer while loop, we must also have $x<100$. The formula at the end of the program fragment at hand states that the value of x is the prime of rank ν_1. All the other properties regarding array a still hold. We now move on to the statement $y:=x*x$. The precondition of this statement is the postcondition of the triple PRIME-2. Simply propagating the formula using the (ASSIGN) rule, we get the triple PRIME-3.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0))^*\}$$

PRIME-3

$$y := x * x$$

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0^2))^*\}$$

The PRIME-2 and PRIME-3 triples can now be combined with the (SEQ) rule, to produce the triple PRIME-4.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1-1,x-1)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\le\delta\le100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0))^*\}$$

`while` $a[x]\neq 1$ `do`

$x:=x+1$

PRIME-4

`endwhile`

$y:=x*x$

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1,x)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\le\delta\le100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ (\forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0)\wedge y=\delta_0^2))^*\}$$

We now turn our attention to the second inner `while` loop. The first program point inside the inner `while` loop has the following properties: the value of x is the prime of rank ν_1, all the elements of the array whose ranks are non-primes are set to 0, all the elements of the array whose ranks are multiple of primes smaller than the value of x have also been set to zero, and ν_2 of the array elements whose ranks are multiples of x greater than x^2 have been set to 0 as well. The first statement in the inner `while` loop is $a:=a[y\mapsto 0]$. We can now write the triple PRIME-5.

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1) \wedge$$
$$(\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge\ (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \delta_1\delta_2 \geq \delta_0 * \delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0 * (\delta_0 + \nu_2))^*\}$$

$a := a[y \mapsto 0]$ PRIME-5

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 \leq \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \delta_1\delta_2 \geq \delta_0 * (\delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = (\delta_0) * (\delta_0 + \nu_2)))^*\}$$

The postcondition of the triple PRIME-5 can be propagated through the next statement. This results in the triple PRIME-6.

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 \leq \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \ \delta_1\delta_2 \geq \delta_0 * (\delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = (\delta_0) * (\delta_0 + \nu_2)))^*\}$$

$y := y + x$ PRIME-6

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge(\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge\ \ (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 \leq \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \ \delta_1\delta_2 \geq \delta_0 * (\delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0 * (\delta_+\nu_2 + 1)))^*\}$$

Triples PRIME-5 and PRIME-6 can now be combined using the (SEQ) rule, resulting in the triple PRIME-7.

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1) \wedge$$
$$(\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge\ (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \delta_1\delta_2 \geq \delta_0 * \delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0 * (\delta_0 + \nu_2))^*\}$$

$a := a[y \mapsto 0]$ PRIME-7

$y := y + x$

$$\{\lambda\langle\nu_1\nu_2\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge(\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge\ (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 \leq \delta_0 * (\delta_0 + \nu_2)$$
$$\wedge\ \delta_1\delta_2 \geq \delta_0 * (\delta_0 + \nu_2)$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0 * (\delta_+\nu_2 + 1)))^*\}$$

Let us look more closely at the postcondition of the PRIME-7 triple. The value of x is still the prime of rank ν_1. Since the second inner loop has been executed a number of times equal to ν_2, ν_2+1 elements of the array a whose ranks are multiples of x larger than or equal to x^2 have been set to 0. The elements of non-prime ranks smaller than x and the elements whose ranks are multiples of primes smaller than x are still set to 0. We can now use the (WHILE) rule to produce a triple for the second `while` loop.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ (\forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0) \wedge y = \delta_0^2))^*\}$$

`while` $y \leq 100$ `do`

$a := a[y \mapsto 0]$ PRIME-8

$y := y + x$

`endwhile`

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ \ (\exists\delta_0.prime\_rank(\nu_1 + 1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0)^*\}$$

First, we notice that the program outputs the value of x right after it exits the first inner while loop. From the postcondition of triple PRIME-8 we infer that the program will output all the primes, in increasing order. We also notice the fact that the star annotation is retained in the postcondition of this triple. This means that we know exactly *whether the loop terminates or not*. To uncover this information, we look at the family of higher order logic formulas

$$prime\_rank(\mu_1, x) \wedge (\exists\delta_0.prime\_rank(\mu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\mu_1 + 1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2 \rightarrow a[\delta_1\delta_2] = 0)$$

where $\mu_1 = 0, 1, 2, \ldots$. Since each of the formulas in the family are satisfiable for some values of x and a, if follows that the loop terminates for each environment

that satisfies the precondition. Next, we can combine the triples PRIME-4 and PRIME-8 using the (SEQ) rule, resulting in the triple PRIME-9.

$$\begin{aligned}\{\lambda\langle\nu_1\rangle\,.\,(\quad & \mathit{prime\_rank}(\nu_1-1,x-1)\wedge(\exists\delta_0.\mathit{prime\_rank}(\nu_1-1,\delta_0)\wedge\delta_0+1<100)\\ & \wedge\ (\forall\delta\,.\,2\le\delta\le 100\wedge\mathit{prime}(\delta)\rightarrow a[\delta]=1)\\ & \wedge\ (\exists\delta_0.\mathit{prime\_rank}(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg\mathit{prime}(\delta)\rightarrow a[\delta]=0)\\ & \qquad\wedge\ \forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2\\ & \qquad\qquad\rightarrow a[\delta_1\delta_2]=0))^*\}\end{aligned}$$

`while` $a[x]\neq 1$ `do`

$x:=x+1$

`endwhile`

PRIME-9

$y:=x*x$

`while` $y\le 100$ `do`

$a:=a[y\mapsto 0]$

$y:=y+x$

`endwhile`

$$\begin{aligned}\{\lambda\langle\nu_1\rangle\,.\,(\quad & \mathit{prime\_rank}(\nu_1,x)\wedge(\exists\delta_0.\mathit{prime\_rank}(\nu_1-1,\delta_0)\wedge\delta_0+1<100)\\ & \wedge\ (\forall\delta\,.\,2\le\delta\le 100\wedge\mathit{prime}(\delta)\rightarrow a[\delta]=1)\\ & \qquad\wedge\ (\exists\delta_0.\mathit{prime\_rank}(\nu_1+1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg\mathit{prime}(\delta)\rightarrow a[\delta]=0)\\ & \qquad\wedge\ \forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2\\ & \qquad\qquad\rightarrow a[\delta_1\delta_2]=0)^*\}\end{aligned}$$

We can propagate the postcondition of the triple PRIME-9 through the statement $x:=x+1$, resulting in the triple PRIME-10.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ \ (\exists\delta_0.prime\_rank(\nu_1 + 1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0)^*\}$$

PRIME-10

$$x := x + 1$$

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1, x - 1) \wedge (\exists\delta_0.prime\_rank(\nu_1 - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall\delta\,.\,2 \leq \delta \leq 100 \wedge prime(\delta) \rightarrow a[\delta] = 1)$$
$$\wedge\ \ (\exists\delta_0.prime\_rank(\nu_1 + 1, \delta_0) \wedge (\forall\delta\,.\,2 \leq \delta < \delta_0^2 \wedge \neg prime(\delta) \rightarrow a[\delta] = 0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1\delta_2] = 0)^*\}$$

Next, we combine the triples PRIME-9 and PRIME-10 using the (SEQ) rule. We obtain the triple PRIME-11.

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1-1,x-1)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\le\delta\le100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ (\exists\delta_0.prime\_rank(\nu_1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0))^*\}$$

`while` $a[x]\neq1$ `do`

$x:=x+1$

`endwhile`

$y:=x*x$

PRIME-11

`while` $y\le100$ `do`

$a:=a[y\mapsto0]$

$y:=y+x$

`endwhile`

$x:=x+1$

$$\{\lambda\langle\nu_1\rangle\,.\,(\quad prime\_rank(\nu_1,x-1)\wedge(\exists\delta_0.prime\_rank(\nu_1-1,\delta_0)\wedge\delta_0+1<100)$$
$$\wedge\ (\forall\delta\,.\,2\le\delta\le100\wedge prime(\delta)\rightarrow a[\delta]=1)$$
$$\wedge\ \ (\exists\delta_0.prime\_rank(\nu_1+1,\delta_0)\wedge(\forall\delta\,.\,2\le\delta<\delta_0^2\wedge\neg prime(\delta)\rightarrow a[\delta]=0)$$
$$\wedge\ \ \forall\delta_1\,.\,\forall\delta_2\,.\,2\le\delta_1<\delta_0^2\wedge\delta_1\delta_2\ge\delta_0^2$$
$$\rightarrow a[\delta_1\delta_2]=0)^*\}$$

The triple PRIME-11 describes every run through the body of the outer `while` loop. We can now get a grasp of the effect of this body on the variables x and a. At the beginning of this program fragment, x is one more than the prime of rank ν_1-1, and we also have that all array elements whose indices are either non-primes and are smaller than x, or are multiples of primes smaller than x are set to 0. All the other array elements are set to 1. At the end of the the program fragment, x is one more

than the prime of rank ν_1, and we also have that all array elements whose indices are either non-primes and are smaller than x, or are multiples of primes smaller than x are set to 0. We notice that what holds in terms of nu_1 at the beginning of the program fragment, holds in terms of $\nu_1 + 1$ at the end of the program fragment. This provides an intuition of why the (WHILE) loop can be applied next, to produce the PRIME-12 triple.

$\{\lambda\langle\rangle \,.\, (\forall \delta \,.\, a[\delta] = 1 \wedge x = 2)^*\}$

`while` $x < 100$ `do`

  `while` $a[x] \neq 1$ `do`

    $x := x + 1$

  `endwhile`

  $y := x * x$

  `while` $y \leq 100$ `do`

    $a := a[y \mapsto 0]$

    $y := y + x$

  `endwhile`

  $x := x + 1$

`endwhile`

PRIME-12

$$\{\lambda\langle\rangle \,.\, (\exists \delta \,.\quad x \geq 100 \wedge \mathit{prime\_rank}(\delta, x-1) \wedge (\exists \delta_0 . \mathit{prime\_rank}(\delta - 1, \delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall \delta' \,.\, 2 \leq \delta' \leq 100 \wedge \mathit{prime}(\delta') \rightarrow a[\delta'] = 1)$$
$$\wedge\ \ (\exists \delta_0 . \mathit{prime\_rank}(\delta + 1, \delta_0) \wedge (\forall \delta' \,.\, 2 \leq \delta' < \delta_0^2 \wedge \neg \mathit{prime}(\delta') \rightarrow a[\delta'] = 0)$$
$$\wedge\ \ \forall \delta_1 \,.\, \forall \delta_2 \,.\, 2 \leq \delta_1 < \delta_0^2 \wedge \delta_1 \delta_2 \geq \delta_0^2$$
$$\rightarrow a[\delta_1 \delta_2] = 0)^*\}$$

We now have a proof for the entire program given in Figure 13.5. We notice, yet again, that the postcondition of triple PRIME-12 has a star annotation. Since the

higher order formula

$$\exists \delta \,.x \geq 100 \wedge \mathit{prime\_rank}(\delta, x-1) \wedge (\exists \delta_0.\mathit{prime\_rank}(\delta-1,\delta_0) \wedge \delta_0 + 1 < 100)$$
$$\wedge\ (\forall \delta' \,.2 \leq \delta' \leq 100 \wedge \mathit{prime}(\delta') \rightarrow a[\delta'] = 1)$$
$$\wedge\ (\exists \delta_0.\mathit{prime\_rank}(\delta+1,\delta_0) \wedge (\forall \delta' \,.2 \leq \delta' < \delta_0^2 \wedge \neg \mathit{prime}(\delta') \rightarrow a[\delta'] = 0)$$
$$\wedge\ \forall \delta_1 \,.\forall \delta_2 \,.2 \leq \delta_1 < \delta_0^2 \wedge \delta_1\delta_2 \geq \delta_0^2 \rightarrow a[\delta_1\delta_2] = 0)$$

is satisfiable for some environment assigning values to x and a, it follows that the program terminates.

Chapter 14

Symbolic Configurations

In this chapter we continue the development of our framework by introducing a generic algorithm for propagating family description formulas attached to program points. The algorithm consists of a sequence of applications of a transform that computes for each formula attached to a program point its "effect" at the adjoining program points. The "effect" of a formula is defined in a manner similar to the strongest postcondition propagation operator used in classic program verification.

Following the presentation style we used so far in the thesis, we introduce symbolic configurations as means of attaching family description formulas to program point, and then we define our transform in a syntax based manner, as an operator $\mathcal{T}$ mapping from symbolic configurations to symbolic configurations.

The intended use of a symbolic configuration is to allow the inference of program properties; in this respect we need to define an approximation relationship between the symbolic configuration at hand, and a (set of) progressions of interest. Given a symbolic configuration $\mathcal{K}$, such an approximation can be readily defined as $[\![\mathcal{K}]\!]$, which is the family configuration whose annotations are the interpretations of the corresponding formulas appearing in $\mathcal{K}$. In this context, it would be useful

if the symbolic transform could play a role for symbolic configurations similar to the role played by the $\widehat{T}$ operator for family configurations. We will show that this relationship exists if the family description language is well-defined. Then, the use of the symbolic transform $\mathcal{T}$ is twofold: it can be used, on one hand, for checking the correctness of a symbolic configuration, and on the other hand, for refining a given correct configuration, by applying a sequence of propagation steps to it. The refinement process opens the door to automating program reasoning. Indeed, for every program, there exists a default correct symbolic configuration, which has the formula $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, true$ attached to every program point. Applying a sequence of propagation steps to the default correct symbolic configuration would produce correct information about the program without any user input, in a manner similar to program analysis. However, unlike program analysis, this process is incremental, since each step produces correct, usable information, and we do not need to wait until the completion of the algorithm before using its output. On the other hand, a completely automated process is not guaranteed to produce "interesting" information.

Therefore, the reasoning process realized by the algorithm described above could benefit greatly if it could be combined with alternative means of reasoning, like user-specified assertions, and program analysis algorithms. Using assertions in a propagation-based reasoning framework is the object of the next chapter.

14.1 A Strongest Postcondition Operator

We proceed now with the formalization. A configuration whose annotations are formulas is called a *symbolic configuration*. Symbolic configurations are denoted by the symbol $\mathcal{K}$, possibly subscripted. The interpretation of a symbolic configuration

$$\mathcal{T}\left(\begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \mathcal{F}_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \quad \texttt{skip} \\ \langle l_f, \mathcal{F}_s\rangle\end{array} \qquad \text{(PROPAG-SKIP)}$$

$$\mathcal{T}\left(\begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \quad x := E \\ \langle l_f, \mathcal{F}_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \quad x := E \\ \langle l_f, \mathit{Assign}(x, e, \mathcal{F}_s)\rangle\end{array} \qquad \text{(PROPAG-ASSIGN)}$$

$$\mathcal{T}\left(\mathcal{K}_1 \fatsemi \mathcal{K}_2\right) = \mathcal{T}\left(\mathcal{K}_1\right) \fatsemi \mathcal{T}\left(\mathcal{K}_2\right) \qquad \text{(PROPAG-SEQ)}$$

$$\mathcal{T}\left(\begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \quad\quad \mathcal{K}_1 \\ \quad \texttt{else} \\ \quad\quad \mathcal{K}_2 \\ \texttt{endif} \\ \langle l_f, \mathcal{F}_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \texttt{if } C \\ \quad \texttt{then} \\ \quad\quad \langle l_{1s}, \mathit{Filter}(C, \mathcal{K}_1|_{l_{1s}})\rangle \fatsemi \mathcal{T}(\mathcal{K}_1) \\ \quad \texttt{else} \\ \quad\quad \langle l_{2s}, \mathit{Filter}(\neg C, \mathcal{K}_2|_{l_{2s}})\rangle \fatsemi \mathcal{T}(\mathcal{K}_2) \\ \texttt{endif} \\ \langle l_f, \mathcal{K}_1|_{l_{1f}} \sqcup \mathcal{K}_2|_{l_{2f}}\rangle\end{array} \qquad \text{(PROPAG-IF)}$$

where $l_{is} = \mathit{first}(\mathcal{K}_i),\ l_{if} = \mathit{last}(\mathcal{K}_i),\ i \in \{1, 2\}$

$$\mathcal{T}\left(\begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \mathcal{K} \\ \texttt{endwhile} \\ \langle l_f, \mathcal{F}_f\rangle\end{array}\right) = \begin{array}{l}\langle l_s, \mathcal{F}_s\rangle \\ \texttt{while } C \texttt{ do} \\ \quad \langle l'_s, \mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, \mathcal{K}|_{l'_f}))\rangle \fatsemi \widehat{T}(\mathcal{K}) \\ \texttt{endwhile} \\ \langle l_f, \mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}|_{l'_f}))\rangle\end{array} \qquad \text{(PROPAG-WHILE)}$$

where $l_s = \mathit{first}(\mathcal{K}),\ l_f = \mathit{last}(\mathcal{K})$

Figure 14.1: Symbolic Propagation Operator

$\langle 1, \lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*\rangle$
`while` $x > 1$ `do`
  $\langle 2, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1)^*\rangle$
  `if` $x \,\%\, 2 = 0$ `then`
    $\langle 3, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 = 0)^*\rangle$
    $x := x/2$
    $\langle 4, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x) \wedge x > 0)^*\rangle$
  `else`
    $\langle 5, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge a \,\%\, 2 \neq 0)^*\rangle$
    $x := x + 1$
    $\langle 6, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x) \wedge x > 2 \wedge x \,\%\, 2 = 0)^*\rangle$
  `endif`
  $\langle 7, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*\rangle$
`endwhile`
$\langle 8, \lambda\langle\nu_0\rangle\,.\,(x = 1)^*\rangle$

Figure 14.2: Example of Symbolic Configuration

$\mathcal{K}$ is denoted $[\![\mathcal{K}]\!]$. That is, $[\![\mathcal{K}]\!]$ is a family configuration $\hat{K}$ such that $\hat{K}|_l = [\![\mathcal{K}|_l]\!]$, for all labels $l \in \mathit{labels}(\mathcal{K})$. The *strongest postcondition operator* is defined in Figure 14.1. The definition relies on a well-defined family description language, as defined in Definition 12.1. The next proposition shows that the definition of $\mathcal{T}$ is sound, in the sense that $\mathcal{T}$ plays the same role for symbolic configurations as $\widehat{T}$ does for family configurations.

14.1 Proposition Let $\mathcal{K}$ be a symbolic configuration over a well-defined language $\mathcal{L}$. Then $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$.

Proof: Relegated to Appendix A, on page 329.

$\langle 1, \lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*\rangle$
`while` $x > 1$ `do`
$\langle 2, \mathit{Filter}(x > 1, \mathit{Before}(\lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*))\rangle$
`if` $x \,\%\, 2 = 0$ `then`
$\langle 3, \mathit{Filter}(x \,\%\, 2 = 0, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1)^*)\rangle$
$x := x/2$
$\langle 4, \mathit{Assign}(x, x/2, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 = 0)^*)\rangle$
`else`
$\langle 5, \mathit{Filter}(x \,\%\, 2 \neq 0, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1)^*)\rangle$
$x := x + 1$
$\langle 6, \mathit{Assign}(x, x + 1, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 \neq 0)^*)\rangle$
`endif`
$\langle 7, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^* \;\sqcup\; \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*\rangle$
`endwhile`
$\langle 8, \mathit{Filter}(x \leq 1, \lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^* \;\sqcup\; \mathit{Collect}(\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*)\rangle$

Figure 14.3: Application of Stronger Postcondition Operator

14.2 Correctness

We extend the $\vdash$ symbol to symbolic configurations. Given two configurations $\mathcal{K}_1$ and $\mathcal{K}_2$, such that $|\mathcal{K}_1| = |\mathcal{K}_2|$, we write $\mathcal{K}_1 \vdash \mathcal{K}_2$ if $\mathcal{K}_1|_l \vdash \mathcal{K}_2|_l$, for all labels $l \in \mathit{labels}(\mathcal{K}_1)$. Given a program P, and a symbolic configuration $\mathcal{K}$ such that $|\mathcal{K}| = P$, we say that a symbolic configuration is correct if $[\![\mathcal{K}]\!]$ approximates all progressions K such that $\mathcal{K}|_{\mathit{first}(P)} \in [\![\mathcal{K}]\!]|_{\mathit{first}(P)}$. The next lemma gives a sufficient condition for correctness.

14.2 Lemma Let P be a labeled program, and $\mathcal{K}$ a symbolic configuration such that $|\mathcal{K}| = P$ and $\mathcal{T}(\mathcal{K}) \vdash \mathcal{K}$. Then, $\mathcal{K}$ is correct.

Proof: The condition $\mathcal{T}(\mathcal{K}) \vdash \mathcal{K}$ entails that $[\![\mathcal{T}(\mathcal{K})]\!] \preccurlyeq [\![\mathcal{K}]\!]$. From Proposition 14.1

$$
\begin{array}{lllll}
(1) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1^* & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2^* & = \lambda\langle\nu_1\cdots\nu_k\rangle.(\varphi_1\wedge\varphi_2)^*, \\
 & & & & \quad \text{if } \models \forall((\varphi_1\rightarrow\varphi_2)\vee(\varphi_2\rightarrow\varphi_1)) \\
(2) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1^* & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2^* & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1\wedge\varphi_2, \\
 & & & & \quad \text{if } \models \neg\forall((\varphi_1\rightarrow\varphi_2)\vee(\varphi_2\rightarrow\varphi_1)) \\
(3) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1 & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2 & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1\wedge\varphi_2 \\
(4) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1^* & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2 & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1^*, \\
 & & & & \quad \text{if } \models \forall(\varphi_1\rightarrow\varphi_2) \\
(5) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1 & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2^* & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2^*, \\
 & & & & \quad \text{if } \models \forall(\varphi_2\rightarrow\varphi_1) \\
(6) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1^* & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2 & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1\wedge\varphi_2, \\
 & & & & \quad \text{if } \models \neg\forall(\varphi_1\rightarrow\varphi_2) \\
(7) & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_1 & \sqcap & \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi_2^* & = \lambda\langle\nu_1\cdots\nu_k\rangle.\varphi\wedge\varphi_2 \\
 & & & & \quad \text{if } \models \neg\forall(\varphi_2\rightarrow\varphi_1)
\end{array}
$$

Figure 14.4: Definition of the $\sqcap$ operator for LAL* Formulas

it follows that $\widehat{T}([\![\mathcal{K}]\!]) \preccurlyeq [\![\mathcal{K}]\!]$. According to Theorem 11.13, $[\![\mathcal{K}]\!]$ is an approximation of some progression K, such that $K|_l \in [\![\mathcal{K}]\!]|_l$. $\square$

Lemma 14.2 provides a way of checking whether a symbolic configuration is correct. The typical way of doing that is to feed a relevant set of formulas of the form $\mathcal{T}(\mathcal{K})|_l \vdash \mathcal{K}|_l$, where $l \in \mathit{labels}(\mathcal{K})$ into a theorem prover like HOL or PVS [Age92, OSR95]. This process will be described in more detail in the next chapter, where we also consider the use of assertions.

14.3 Remark Let $\mathcal{K}$ be a correct symbolic configuration. Then, $[\![\mathcal{K}]\!]$ must be a superset of a fixpoint of $\widehat{T}$. It follows that $\widehat{T}([\![\mathcal{K}]\!])$ is a superset of a fixpoint of $\widehat{T}$. According to Proposition 14.1, $\widehat{T}([\![\mathcal{K}]\!]) \hat{\preccurlyeq} [\![\mathcal{T}(\mathcal{K})]\!]$, which entails that $\mathcal{T}(\mathcal{K})$ is correct. Reasoning inductively, it follows that $\mathcal{T}^k(\mathcal{K})$ is correct, for all $k \geq 0$. $\square$

$\langle 1, \lambda\langle\rangle \,.\, x \,\%\, 2 = 0 \wedge y \,\%\, 2 = 0\rangle$
while $x < n$ **do**
$\langle 2, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
if $x \,\%\, 2 = 0$ **then**
$\langle 3, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
$y := y + 1$
$\langle 4, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
else
$\langle 5, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
$y := y - 1$
$\langle 6, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
endif
$\langle 7, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \rightarrow y \,\%\, 2 = 1\rangle$
$x := x + 1$
$\langle 8, \lambda\langle\nu\rangle \,.\, \mathit{true}\rangle$
endwhile
$\langle 9, \lambda\langle\rangle \,.\, \mathit{true}\rangle$

Figure 14.5: Configuration Before Being Subjected to Propagation

The rest of this section is devoted to an example of applying the strongest postcondition operator to a symbolic configuration and proving it correct. In this example we shall use the program given in Figure 13.3, for which we have given a progressive Hoare proof that showed termination in Section 13.4. Figure 14.2 shows a correct symbolic configuration for this program, and Figure 14.3 shows the result of applying the strongest postcondition operator to this configuration. Let us denote by $\mathcal{K}$ the configuration in Figure 14.2. Then, the configuration given in Figure 14.3 is $\mathcal{T}(\mathcal{K})$. We shall prove the correctness of $\mathcal{K}$ by showing that $\mathcal{T}(\mathcal{K}) \vdash \mathcal{K}$, which in fact means that $\mathcal{T}(\mathcal{K})|_l \vdash \mathcal{K}|_l$, for all $l \in \mathit{labels}(\mathcal{K})$. This translates into proving that the following formulas are theorems of LAL*.

1. $\lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^* \vdash \lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*$

2. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1)^* \vdash$

$$\mathit{Filter}(x > 1, \mathit{Before}(\lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^*, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*))$$

3. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 = 0)^* \vdash$

$$\mathit{Filter}(x \,\%\, 2 = 0, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 0)^*)$$

4. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x) \wedge x > 1)^* \vdash$

$$\mathit{Assign}(x, x/2, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 = 0)^*)$$

5. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 \neq 0)^* \vdash$

$$\mathit{Filter}(x \,\%\, 2 \neq 0, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1)^*)$$

6. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^* \vdash$

$$\mathit{Assign}(x, x + 1, \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 \neq 0)^*)$$

7. $\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^* \vdash$

$$\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^* \;\sqcup\; \lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*$$

8. $\lambda\langle\nu_0\rangle\,.\,(x = 1)^* \vdash$

$$\mathit{Filter}(x \leq 1, \lambda\langle\nu_0\rangle\,.\,(B(\nu_0, x))^* \;\sqcup\; \mathit{Collect}(\lambda\langle\nu_0\nu_1\rangle\,.\,(B(\nu_0 - \nu_1 - 1, x))^*)$$

The formulas are numbered by their labels in the program. Applying the definitions of the *Assign*, *Filter*, *Before*, *Collect* and $\sqcup$ given in Section 13.1, and the LAL* axioms given in Figure 13.2, we get the following first order proof obligations.

1. $B(\nu_0, x) \rightarrow B(\nu_0, x)$

2. $B(\nu_0 - \nu_1, x) \wedge x > 1 \rightarrow x > 1 \wedge (\nu_1 = 0 \rightarrow B(\nu_0, x)) \wedge (\nu_1 > 0 \rightarrow B(\nu_0 - \nu_1, x))$

3. $B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 = 0 \rightarrow x \,\%\, 2 = 0 \wedge B(\nu_0 - \nu_1, x) \wedge x > 1$

4. $B(\nu_0 - \nu_1 - 1, x) \wedge x > 0 \rightarrow B(\nu_0 - \nu_1, 2 * x) \wedge 2 * x > 1 \wedge 2 * x \,\%\, 2 = 0$

5. $B(\nu_0 - \nu_1, x) \wedge x > 1 \wedge x \,\%\, 2 \neq 0 \rightarrow x \,\%\, 2 \neq 0 \wedge B(\nu_0 - \nu_1, x) \wedge x > 1$

6. $B(\nu_0 - \nu_1 - 1, x) \wedge x > 2 \wedge x \,\%\, 2 = 0 \rightarrow$
$$B(\nu_0 - \nu_1, x - 1) \wedge x - 1 > 1 \wedge x - 1 \,\%\, 2 \neq 0$$

7. $B(\nu_0 - \nu_1 - 1, x) \rightarrow B(\nu_0 - \nu_1 - 1, x) \ \vee \ B(\nu_0 - \nu_1 - 1, x)$

8. $x = 1 \rightarrow x \leq 1 \wedge B(\nu_0, x) \ \vee \ \exists \delta \,.\, B(\nu_0 - \delta - 1, x)$

The proofs of theorems 1,2,3,5,7, and 8 are immediate. In order to prove theorems 4 and 6, we note that $\forall x\, \delta \,.\, B(\delta - 1, x) \rightarrow B(\delta, 2 * x)$ and $\forall x\, \delta \,.\, x \,\%\, 2 = 0 \rightarrow B(\delta - 1, x) \rightarrow B(\delta, x - 1)$ are first-order theorems.

14.3 Propagation

When reasoning about programs, it is often useful to compute the effect of a formula annotation being correct, while it may not yet be known whether the formula is indeed correct. The concept behind such computation is called *propagation*, and in our setting, when applied to a symbolic configuration $\mathcal{K}$, it produces a symbolic configuration $\mathcal{K}'$ with the property that if $\mathcal{K}'$ is correct, then so is $\mathcal{K}$. In our framework, propagation is based on a propagation operator, which works according to the principle given in Remark 2.7. In other words, given a monotone operator T, we have a monotone and *decreasing* operator in $T \sqcap I$. Thus, on a lattice $(L, \subseteq)$, where $\subseteq$ denotes an approximation relation, if we have a set X and we would like to show that X approximates the least fixpoint of an operator T, then we can proceed in the following way. First we check if $T(X) \subseteq X$. If that is true, then X is indeed an approximation of $\mathit{lfp}(T)$. However, if $T(X) \not\subseteq X$, then we compute an element

$\langle 1, \lambda\langle\rangle \,.\, x \,\%\, 2 = 0 \wedge y \,\%\, 2 = 0\rangle$
while $x < n$ **do**
$\langle 2, \lambda\langle\nu\rangle \,.\, true\rangle$
if $x \,\%\, 2 = 0$ **then**
$\langle 3, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0\rangle$
$y := y + 1$
$\langle 4, \lambda\langle\nu\rangle \,.\, true\rangle$
else
$\langle 5, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 \neq 0\rangle$
$y := y - 1$
$\langle 6, \lambda\langle\nu\rangle \,.\, true\rangle$
endif
$\langle 7, \lambda\langle\nu\rangle \,.\, true\rangle$
$x := x + 1$
$\langle 8, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 1 \rightarrow y \,\%\, 2 = 1\rangle$
endwhile
$\langle 9, \lambda\langle\rangle \,.\, true\rangle$

Figure 14.6: Application of Strongest Postcondition Operator

$Y = (T \cap I)^k(X)$ for some $k > 0$, and then check whether $T(Y) \subseteq Y$. If that happens, then Y is an approximation of $lfp(T)$, and so is X. In our case, the role of the T operator is played by the strongest postcondition operator $\mathcal{T}$. What is missing from the picture is the *meet operator*. In this section we show how a meet operator can be defined, and we define such an operator for the LAL* formal system. Then, we present an example where we compute a provably correct symbolic configuration using propagation.

The meet operator for formulas is denoted by $\sqcap$ and must be defined as an operator that computes the greatest lower bound of two formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, with the same arity. More specifically, for all formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ of a language $\mathcal{L}$,

$\mathcal{F}_1 \vdash \mathcal{F}_1 \sqcap \mathcal{F}_2$ and $\mathcal{F}_2 \vdash \mathcal{F}_1 \sqcap \mathcal{F}_2$ are theorems in the corresponding formal system $\mathcal{L}^*$. Moreover, for all formulas $\mathcal{F}$ such that $\mathcal{F} \vdash \mathcal{F}_1$ and $\mathcal{F} \vdash \mathcal{F}_2$ are theorems of $\mathcal{L}^*$, we have that $\mathcal{F} \vdash \mathcal{F}_1 \sqcap \mathcal{F}_2$ is also a theorem.

Figure 14.4 shows the definition of the $\sqcap$ meta-operator for the LAL* formal system. In this definition, φ_1 and φ_2 are simple formulas, $k \geq 0$ is a natural number, and $\nu_1, \ldots, \nu_k$ are index variables that appear free in φ_1 and φ_2. The definition is made up of seven cases. The first case sets the rule to be used with starred formulas. In this case, the star annotation can be retained only if $\varphi_1 \rightarrow \varphi_2$ or $\varphi_2 \rightarrow \varphi_1$ are first-order theorems. If this is not the case, then the interpretation of the intersection may contain the empty set, in which case, the star annotation cannot be retained. This is expressed in the second case of the definition. Case (3) sets the rule for non-starred formulas. Cases (4), (5), (6) and (7) define the $\sqcap$ operator for mixed operands. It is interesting to note that whenever the starred formula implies the non-starred formula, the star annotation can be retained.

14.4 Remark Following Remark 14.3, given a correct symbolic configuration $\mathcal{K}$, since $\mathcal{T}(\mathcal{K})$ is correct, it follows that $\mathcal{K} \sqcap \mathcal{T}(\mathcal{K})$ is correct, that is, $(\mathcal{T} \sqcap \mathcal{I})(\mathcal{K})$ is correct. Reasoning inductively, it follows that $(\mathcal{T} \sqcap \mathcal{I})^k(\mathcal{K})$ is correct for all $k \geq 0$. □

Figures 14.5, 14.6, and 14.7 present an example of propagation. In Figure 14.5 we present a configuration for a program which has a while loop that increments a counter x, and operates on the variable y in such a way that the parities of x and y are synchronized. Except for program point 7, the annotations attached to all program points are $\lambda\langle\nu\rangle\,.\,\mathit{true}$. Since $\lambda\langle\nu\rangle\,.\,\varphi \vdash \lambda\langle\nu\rangle\,.\,\mathit{true}$ is a theorem of LAL* for all simple formulas φ, the annotations are surely correct. For program point 7, we have the annotation $\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \rightarrow y \,\%\, 2 = 1 \wedge x < n$, which

$\langle 1, \lambda\langle\rangle \,.\, x \,\%\, 2 = 0 \wedge y \,\%\, 2 = 0\rangle$
while $x < n$ **do**
$\langle 2, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1 \wedge x < n\rangle$
if $x \,\%\, 2 = 0$ **then**
$\langle 3, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \wedge y \,\%\, 2 = 0 \wedge x < n\rangle$
$y := y + 1$
$\langle 4, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \wedge y \,\%\, 2 = 1 \wedge x < n\rangle$
else
$\langle 5, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 \neq 0 \wedge y \,\%\, 2 = 1 \wedge x < n\rangle$
$y := y - 1$
$\langle 6, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 1 \wedge y \,\%\, 2 = 0 \wedge x < n\rangle$
endif
$\langle 7, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1 \wedge x < n\rangle$
$x := x + 1$
$\langle 8, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1 \wedge x < n + 1\rangle$
endwhile
$\langle 9, \lambda\langle\rangle \,.\, x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1 \wedge x = n\rangle$

Figure 14.7: Example of Propagation

expresses only half of the synchrony between the two variables (by replacing the implication with an equivalence, we would get both halves). Let us denote by $\mathcal{K}$ the configuration in Figure 14.5. While being correct, $\mathcal{K}$ cannot be proved to be correct. Figure 14.6 illustrates this fact by showing the configuration $\mathcal{T}(\mathcal{K})$. To see that $\mathcal{T}(\mathcal{K}) \not\vdash \mathcal{K}$, we can look at program point (7) in the two figures: clearly $\lambda\langle\nu\rangle \,.\, \mathit{true} \not\vdash \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \rightarrow y \,\%\, 2 = 1 \wedge x < n$. Let us denote by $\mathcal{I}$ the identity operator for symbolic families. Figure 14.7 shows the symbolic configuration $\mathcal{K}' = (\mathcal{T} \sqcap \mathcal{I})^{12}(\mathcal{K})$. It can be easily verified that $\mathcal{K}'$ is a fixpoint of $\mathcal{T}$, and therefore provably correct. Since $\mathcal{K}' \vdash \mathcal{K}$, if follows that $\mathcal{K}$ is correct as well.

Chapter 15

Conditional Reasoning

As argued in the introduction, the desirable features of a program reasoning framework are the ability to express a wide range of program properties, compositionality, as well as the ability to use assertions, and incrementality. By introducing the progressive semantics and defining a Hoare-like progressive calculus, we have created the grounds for a compositional program reasoning framework. Moreover, the liveness-aware language introduced in Chapter 13 allows for expressing safety, liveness and sequence-based properties in a unified manner. It is now time to complete the picture by introducing an incremental program reasoning methodology that uses propagation and user-provided assertions to verify program properties. Assertions are simply formulas attached to program points whose role is twofold: on one hand, we can see the effect of an assertion being correct propagated throughout the entire program; on the other hand, using the result given in Proposition 14.2, we may be able to show that assertions are correct, and thus be able to guarantee that certain properties of interest hold. In order to ensure incrementality, instead of proving assertions, we prove conditional relationships between assertions, with the meaning that the correctness of an assertion depends on the correctness of a set of some

other assertions whose proof may be attempted later. The concept behind building conditional relationships is called *conditional correctness*, and is the topic discussed in this chapter.

15.1 Assertions

In this section we introduce assertions, which are formula of the form $assert_{\{l\}}(\mathcal{F})$, where l is a label, and $\mathcal{F}$ is a family description language formula. Intuitively, an assertion states that the formula $\mathcal{F}$ is correct at program point with label l. Using the propagation operator, we may compute the effect of the assertion being true. As a result of propagation formulas of the form $assert_{\{l\}}(\mathcal{F}')$ may appear at other program point, say with label l'. Then $\mathcal{F}'$ can be interpreted as a property that is correct at program point l' if $\mathcal{F}$ is correct at l. Moreover, in the process of reasoning, we may be able to prove that $\mathcal{F}$ is conditionally correct on the assertions at program points $l_1, \ldots, l_k$. In this case, we can replace the $assert_{\{l\}}(\cdots)$ construct by $assert_{\{l_1,\ldots,l_k\}}(\cdots)$ throughout the entire program. The intended outcome of the reasoning process is to turn the $\{l_1, \ldots, l_k\}$ sets of labels into the empty set, that is, to prove all assertions unconditionally. However, useful information can be derived before the completion of the reasoning process, in the form of conditionally correct assertions. We now proceed with the formalization.

Given a family description language $\mathcal{L}$ and a program P, the *assertion language* $\mathcal{L}_P$ *based on* $\mathcal{L}$ *for* P is defined as follows. If $\mathcal{F}$ is a formula in $\mathcal{L}$, then $assert_{\Lambda}(\mathcal{F})$ is a formula in $\mathcal{L}_P$, where $\Lambda \subseteq labels(P)$. If $\mathcal{F}$ and $\mathcal{F}'$ are formulas in $\mathcal{L}_P$, and x is a program variable, E is a program expression, and C is a program constraint, then $\mathcal{F} \sqcap \mathcal{F}'$, $Assign(x, E, \mathcal{F})$, $Filter(C, \mathcal{F})$, $Before(\mathcal{F}, \mathcal{F}')$, $Collect(\mathcal{F})$ and $\mathcal{F} \sqcup \mathcal{F}'$ are formulas in $\mathcal{L}_P$. We call formulas in $\mathcal{L}_P$ *assertions*. In order to manipulate

assertions, we need to be able to turn them "on" and "off" selectively. This can be achieved with the help of the *assume* operator. Given a program P, and a family description language $\mathcal{L}$, the *assume* operator is defined recursively as follows.

a) $assume_{\Lambda}(\mathcal{F}) = \mathcal{F}$ where $\mathcal{F} \in \mathcal{L}$, and $\Lambda \subseteq labels(P)$.

b) $assume_{\Lambda_1}(assert_{\Lambda_2}(\mathcal{F})) = assume_{\Lambda_1}(\mathcal{F})$, if $\Lambda_2 \subseteq \Lambda_1$, where $\Lambda_1 \subseteq labels(P)$, and $\mathcal{F} \in \mathcal{L}$.

c) $assume_{\Lambda_1}(assert_{\Lambda_2}(\mathcal{F})) = \lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,true$, if $\Lambda_2 \backslash \Lambda_1 \neq \emptyset$, where $\Lambda_1 \subseteq labels(P)$, $\Lambda_2 \subseteq labels(P)$ and $\mathcal{F} \in \mathcal{L}$ such that $arity(\mathcal{F}) = k$.

d) $assume_{\Lambda}(\mathcal{F} \sqcap \mathcal{F}') = assume_{\Lambda}(\mathcal{F}) \sqcap assume_{\Lambda}(\mathcal{F}')$, where $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

e) $assume_{\Lambda}(Assign(x, E, \mathcal{F})) = Assign(x, E, assume_{\Lambda}(\mathcal{F}))$, where x is a program variable, E is a program expression, $\mathcal{F} \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

f) $assume_{\Lambda}(Filter(C, \mathcal{F})) = Filter(C, assume_{\Lambda}(\mathcal{F}))$, where C is a program constraint, $\mathcal{F} \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

g) $assume_{\Lambda}(Before(\mathcal{F}, \mathcal{F}')) = Before(assume_{\Lambda}(\mathcal{F}), assume_{\Lambda}(\mathcal{F}'))$, where $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

h) $assume_{\Lambda}(Collect(\mathcal{F})) = Collect(assume_{\Lambda}(\mathcal{F}))$, $\mathcal{F} \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

i) $assume_{\Lambda}(\mathcal{F} \sqcup \mathcal{F}') = assume_{\Lambda}(\mathcal{F}) \sqcup assume_{\Lambda}(\mathcal{F}')$, where $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$, and $\Lambda \subseteq labels(P)$.

15.1 Remark For all symbolic configurations $\mathcal{K}$ and sets of labels Λ_1 and Λ_2, such that $\Lambda_1 \subseteq labels(\mathcal{K})$ and $\Lambda_2 \subseteq labels(\mathcal{K})$, we have that $assume_{\Lambda_1 \cup \Lambda_2}(\mathcal{K}) = assume_{\Lambda_1}(\mathcal{K}) \sqcap assume_{\Lambda_2}(\mathcal{K})$. □

A *verification problem* is a symbolic configuration $\mathcal{K}$ such that $\mathcal{K}|_l$ is either of the form $assert_{\{l\}}(\mathcal{F})$, for some $\mathcal{F} \in \mathcal{L}$, or $\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,true$, where k is a suitably chosen arity. We extend the *assume* notation to verification problems. Given a verification problem $\mathcal{K}$, and a set of labels $\Lambda \subseteq labels(\mathcal{K})$, $assume_\Lambda(\mathcal{K})$ is a symbolic configuration $\mathcal{K}'$ such that $|\mathcal{K}| = |\mathcal{K}'|$, and $\mathcal{K}'|_l = assume_\Lambda(\mathcal{K}|_l)$.

15.2 Proposition Let $\mathcal{K}$ be a verification problem and Λ_1, Λ_2 two sets of labels such that $\Lambda_1 \subseteq \Lambda_2 \subseteq labels(\mathcal{K})$. Then, $assume_{\Lambda_2}(\mathcal{K}) \vdash assume_{\Lambda_1}(\mathcal{K})$.

Proof: For all labels $l \in \Lambda_1 \cup (labels(\mathcal{K})\backslash\Lambda_2)$, we have that $assume_{\Lambda_2}(\mathcal{K})|_l = assume_{\Lambda_1}(\mathcal{K})|_l$, so then $assume_{\Lambda_2}(\mathcal{K})|_l \vdash assume_{\Lambda_1}(\mathcal{K})|_l$ holds in $\mathcal{L}^*$. For $l \in \Lambda_2\backslash\Lambda_1$, we have that $assume_{\Lambda_1}(\mathcal{K})|_l = \lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,true$, where k is the arity of $assume_{\Lambda_2}(\mathcal{K})|_l$. Since $\mathcal{F} \vdash \lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,true$ holds in $\mathcal{L}^*$ for all formulas $\mathcal{F}$ of arity $k \geq 0$, it follows that $assume_{\Lambda_2}(\mathcal{K})|_l \vdash assume_{\Lambda_1}(\mathcal{K})|_l$ holds. We just proved that $assume_{\Lambda_2}(\mathcal{K})|_l \vdash assume_{\Lambda_1}(\mathcal{K})|_l$ holds for all $l \in labels(\mathcal{K})$, which entails that $assume_{\Lambda_2}(\mathcal{K}) \vdash assume_{\Lambda_1}(\mathcal{K})$. $\square$

Since the $\mathcal{T}$ propagation operator is defined in terms of the *Assign*, *Filter*, *Before*, *Collect* and $\sqcup$ operators, $\mathcal{T}$ can also be applied to verification problems. Next, we define the label replacement operator for assertions. The expression $replace(\mathcal{F}, l, \Lambda)$ denotes the assertion obtained from $\mathcal{F}$ by replacing the label l by the labels in the set Λ. Given a program P and a family description language $\mathcal{L}$, the operator *replace* is defined recursively as follows.

a) $replace(\mathcal{F}, l, \Lambda) = \mathcal{F}$, where $\mathcal{F} \in \mathcal{L}$, $l \in labels(P)$, and $\Lambda \subseteq labels(P)$.

b) $replace(assert_\Lambda(\mathcal{F}), l, \Lambda') = assert_{(\Lambda\cup\Lambda')\backslash\{l\}}(\mathcal{F})$, if $l \in \Lambda$, where $\Lambda \subseteq labels(P)$, $\Lambda' \subseteq labels(P)$, and $\mathcal{F} \in \mathcal{L}$.

c) $replace(assert_{\{l\}}(\mathcal{F}), l, \emptyset) = \mathcal{F}$, where $l \in labels(P)$, and $\mathcal{F} \in \mathcal{L}$.

d) $replace(assert_{\Lambda}(\mathcal{F}), l, \Lambda') = assert_{\Lambda}(\mathcal{F})$, if $l \notin \Lambda$, where $\Lambda \subseteq labels(P)$, $\Lambda' \subseteq labels(P)$, and $\mathcal{F} \in \mathcal{L}$.

e) $replace(\mathcal{F} \sqcap \mathcal{F}', l, \Lambda) = replace(\mathcal{F}, l, \Lambda) \sqcap replace(\mathcal{F}', l, \Lambda)$, where $\Lambda \subseteq labels(P)$, $l \in labels(P)$, and $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$.

f) $replace(Filter(C, \mathcal{F}), l, \Lambda) = Filter(C, replace(\mathcal{F}, l, \Lambda))$, where C is a program constraint, $l \in labels(P)$, $\Lambda \subseteq labels(P)$, and $\mathcal{F} \in \mathcal{L}_P$.

g) $replace(Before(\mathcal{F}, \mathcal{F}'), l, \Lambda) = Before(replace(\mathcal{F}, l, \Lambda), replace(\mathcal{F}', l, \Lambda))$, where $\Lambda \subseteq labels(P)$, $l \in labels(P)$, and $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$.

h) $replace(\mathcal{F} \sqcup \mathcal{F}', l, \Lambda) = replace(\mathcal{F}, l, \Lambda) \sqcup replace(\mathcal{F}', l, \Lambda)$, where $\Lambda \subseteq labels(P)$, $l \in labels(P)$, and $\mathcal{F}, \mathcal{F}' \in \mathcal{L}_P$.

i) $replace(Collect(\mathcal{F}), l, \Lambda) = Collect(replace(\mathcal{F}, l, \Lambda))$, where $\Lambda \subseteq labels(P)$, $l \in labels(P)$, and $\mathcal{F} \in \mathcal{L}_P$.

We also extend the *replace* operator to symbolic configurations, in the obvious way, that is, $replace(\mathcal{K}, l, \Lambda)$ is a symbolic configuration $\mathcal{K}'$ such that $|\mathcal{K}'| = |K|$ and $\mathcal{K}'|_l = replace(\mathcal{K}|_l, l, \Lambda)$.

15.2 Conditional Correctness

In this section, we introduce the concept of conditional correctness and prove a couple of properties that shall contribute to the development of a program reasoning methodology. Before discussing conditional correctness, let us define the (unconditional) correctness of an assertion. Given a verification problem $\mathcal{K}$, we say that assertion at label $l \in labels(\mathcal{K})$ is *(unconditionally) correct* if $\mathcal{T}(assume_{\{l\}}(\mathcal{K}) \vdash$

$assume_{\{l\}}(\mathcal{K})$. Conditional correctness of assertions is defined as a relationship between assertions in such a way that once the relationship is established, the correctness of one assertion entails the correctness of other assertions that are conditionally correct on the first assertion. To provide a better understanding of this concept, let us take a closer look at the role of assertions in program reasoning. The use of assertions has been introduced as a means for the programmer to specify conditions (properties) of the program that are expected to be correct, to help along in the debugging process. In a modular software development environment, program fragments may be developed and tested separately. However, since these program fragments are intended to become parts of the same system, each of the fragments operates under certain preconditions, that should be reflected in the set of start environments for the respective fragment. Therefore, the assertions specified inside the program fragment hold only if upon entry into that program fragment, its precondition held. In other words, the assertion is correct on condition that the precondition be correct. Our definition of conditional correctness implements exactly this relationship.

15.3 Definition Let $\mathcal{K}$ be a verification problem, $l \in labels(\mathcal{K})$ a label, and $\Lambda \subseteq labels(\mathcal{K})$ a set of labels such that $l \notin \Lambda$. We say that assertion at program point l is *conditionally correct* on assertions at program points in Λ if $assume_{\{l\}}(\mathcal{K})$ is correct whenever $assume_{\Lambda}(\mathcal{K})$ is correct. □

In other words, once we have established the conditional correctness of the assertion at program point l on the assertions at program points in a set of labels Λ, only the assertions in Λ remain as proof obligation. Establishing the conditional correctness of an assertion can be seen as an incremental step in the process of proving all the assertions.

$\langle 1, assert_{\{1\}}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 0)\rangle$
while $x < n$ **do**
  $\langle 2, \lambda\langle\nu\rangle \,.\, true\rangle$
  if $x \,\%\, 2 = 0$ **then**
    $\langle 3, \lambda\langle\nu\rangle \,.\, true\rangle$
    $y := y + 1$
    $\langle 4, \lambda\langle\nu\rangle \,.\, true\rangle$
  else
    $\langle 5, \lambda\langle\nu\rangle \,.\, true\rangle$
    $y := y - 1$
    $\langle 6, \lambda\langle\nu\rangle \,.\, true\rangle$
  endif
  $\langle 7, \lambda\langle\nu\rangle \,.\, true\rangle$
  $x := x + 1$
  $\langle 8, assert_{\{8\}}(\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0)\rangle$
endwhile
$\langle 9, \lambda\langle\rangle \,.\, true\rangle$

Figure 15.1: A Verification Problem

We now prepare the ground for the main result of this chapter. The following two propositions shall be useful in proving Theorem 15.6.

15.4 Proposition Let $\mathcal{K}_0$ be a verification problem, $l \in labels(\mathcal{K}_0)$ a label, and $\Lambda \subseteq labels(\mathcal{K}_0)$ a set of labels such that $l \notin \Lambda$. Denote by $\mathcal{K}$ the configuration $(\mathcal{T} \sqcap \mathcal{I})^k(\mathcal{K}_0)$, for some $k \geq 0$. If $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K})) \vdash assume_{\{l\}}(\mathcal{K})$ holds, then assertion at program point l in the original configuration $\mathcal{K}_0$ is conditionally correct on assertions at program points in Λ.

Proof: According to Remark 15.1, we have $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K})) = assume_{\Lambda}(\mathcal{T}(\mathcal{K})) \sqcap assume_{\{l\}}(\mathcal{T}(\mathcal{K}))$. Then, using the hypothesis, we have

that $assume_{\Lambda}(\mathcal{T}(\mathcal{K})) \sqcap assume_{\{l\}}(\mathcal{T}(\mathcal{K})) \vdash assume_{\{l\}}(\mathcal{K})$. Assume now that $assume_{\Lambda}(\mathcal{T}(\mathcal{K}))$ is correct. Using Lemma 14.2, it follows that $assume_{\Lambda}(\mathcal{T}(\mathcal{K})) \vdash assume_{\Lambda}(\mathcal{K})$. Combining the two relations, we get $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K})) \vdash assume_{\Lambda\cup\{l\}}(\mathcal{K})$, which entails that $assume_{\Lambda\cup\{l\}}(\mathcal{K})$ is correct. □

15.5 Proposition Let $\mathcal{K}$ be a symbolic configuration whose annotations contain *assert* constructs whose subscripts may be labels not necessarily in $labels(\mathcal{K})$. Let l and Λ be a label, and a set of labels, respectively, not necessarily from $labels(\mathcal{K})$, with $l \notin \Lambda$. If $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K})|_{l'}) \vdash assume_{\{l\}}(\mathcal{K}|_{l'})$ holds for some label $l' \in labels(\mathcal{K})$, then $assume_{\Lambda\cup\{l\}}(\mathcal{T}((\mathcal{T}\sqcap\mathcal{I})(\mathcal{K}))|_{l'}) \vdash assume_{\{l\}}((\mathcal{T}\sqcap\mathcal{I})(\mathcal{K})|_{l'})$ holds.

Proof: Relegated to Appendix A, on page 331.

The central result of this chapter is captured in the next theorem, which states a simple condition for proving the conditional correctness of an assertion. It turns out that it is sufficient to inspect only the formula annotations attached to a single program point in order to determine the conditional correctness of an assertion.

15.6 Theorem Let $\mathcal{K}_0$ be a verification problem, $l \in labels(\mathcal{K}_0)$ a label, and $\Lambda \subseteq labels(\mathcal{K}_0)$ a set of labels such that $l \notin \Lambda$. Denote by $\mathcal{K}$ the configuration $(\mathcal{T}\sqcap\mathcal{I})^k(\mathcal{K}_0)$, for some $k \geq 0$. If $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K}))|_l \vdash assume_{\{l\}}(\mathcal{K})|_l$, then assertion at program point l in the original configuration $\mathcal{K}_0$ is conditionally correct on assertions at program points in Λ.

Proof: Clearly, $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K}_0))|_{l'} \vdash assume_{\{l\}}(\mathcal{K}_0)|_{l'}$, for all labels $l' \in labels(\mathcal{K})\backslash\{l\}$. According to Proposition 15.5, This property is preserved by the application of propagation steps, that is, $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K}))|_{l'} \vdash assume_{\{l\}}(\mathcal{K})|_{l'}$, for all labels $l' \in labels(\mathcal{K})\backslash\{l\}$. Using the hypothesis, $\mathcal{K}$ also has the property

$assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K}))|_l \vdash assume_{\{l\}}(\mathcal{K})|_l$, and if follows that $assume_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K}))|_{l'} \vdash assume_{\{l\}}(\mathcal{K})|_{l'}$, for all labels $l' \in labels(\mathcal{K})$. We now use Proposition 15.4, to prove the theorem. □

The next theorem shall be useful in establishing a program reasoning methodology. Once the assertion at program point l has been proved conditionally correct on assertions at program points in the set Λ, we can replace l by labels in Λ throughout the entire symbolic configuration, without losing the chance of proving the correctness of all assertions.

15.7 Theorem Let $\mathcal{K}_0$ be a verification problem, $l \in labels(\mathcal{K}_0)$ a label, and $\Lambda \subseteq labels(\mathcal{K}_0)$ a set of labels such that $l \notin \Lambda$. Assume that assertion at program point l is conditionally correct on assertions at program points in Λ. If $assume_{labels(\mathcal{K})\setminus\{l\}}(replace(\mathcal{K}, l, \Lambda))$ is correct, then so is $assume_{labels(\mathcal{K})\setminus\{l\}}(\mathcal{K})$.

Proof: The replacement operation performs simply a syntactic transformation. It is clear that

$$assume_{labels(\mathcal{K})\setminus\{l\}}(replace(\mathcal{K}, l, \Lambda)) = assume_{labels(\mathcal{K})}(\mathcal{K}) \quad (*)$$

Also, given two sets of labels Λ_1 and Λ_2 such that $\Lambda_1 \subseteq \Lambda_2 \subseteq labels(\mathcal{K})$, we have that if assertion at program point l is conditionally correct on assertions at program points in Λ_1, then assertion at program point l is also conditionally correct on assertions at program points in Λ_2. Using the hypothesis, assertion at program point l is conditionally correct on assertions at program points in $labels(\mathcal{K})\setminus\{l\}$. This means that $assume_{labels(\mathcal{K})}$ is correct if $assume_{labels(\mathcal{K})\setminus\{l\}}(\mathcal{K})$ is correct. Using relation $(*)$, the proof follows. □

$\langle 1, \mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 0)\rangle$

`while` $x < n$ `do`

$\langle 2, \mathit{Filter}(x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 0),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0)))\rangle$

`if` $x \,\%\, 2 = 0$ `then`

$\langle 3, \mathit{Filter}(x \,\%\, 2 = 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 0),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

$y := y + 1$

$\langle 4, \mathit{Filter}(x \,\%\, 2 = 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 1),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))\rangle$

`else`

$\langle 5, \mathit{Filter}(x \,\%\, 2 \neq 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 0),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

$y := y - 1$

$\langle 6, \mathit{Filter}(x \,\%\, 2 \neq 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = -1),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))\rangle$

`endif`

$\langle 7, \mathit{Filter}(x \,\%\, 2 = 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 1),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))$

$\sqcup$

$\mathit{Filter}(x \,\%\, 2 \neq 0 \wedge x < n, \mathit{Before}(\mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = -1),$
$\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))\rangle$

$x := x + 1$

$\langle 8, \mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0)\rangle$

`endwhile`

$\langle 9, \mathit{Filter}(x \geq n, \mathit{assert}_{\{1\}}(\lambda\langle\rangle\,.\,x = 0 \wedge y = 0) \sqcup$
$\mathit{Collect}(\mathit{assert}_{\{8\}}(\lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0)))\rangle$

Figure 15.2: Application of Propagation Steps to a Verification Problem

$\langle 1, \lambda\langle\rangle \,.\, x = 0 \wedge y = 0\rangle$

`while` $x < n$ `do`

$\langle 2, \mathit{Filter}(x < n, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 0,$
$\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

`if` $x \,\%\, 2 = 0$ `then`

$\langle 3, \mathit{Filter}(x \,\%\, 2 = 0 \wedge x < n, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 0),$
$\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

$y := y + 1$

$\langle 4, \mathit{Filter}(x \,\%\, 2 = 0 \wedge x < n, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 1,$
$\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))\rangle$

`else`

$\langle 5, \mathit{Filter}(x \,\%\, 2 \neq 0 \wedge x < n, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 0,$
$\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

$y := y - 1$

$\langle 6, \mathit{Filter}(x \,\%\, 2 \neq 0 \wedge x < n, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = -1,$
$\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1))\rangle$

`endif`

$\langle 7, \mathit{Before}(\lambda\langle\rangle \,.\, x = 0 \wedge y = 1, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 1)\rangle$

$x := x + 1$

$\langle 8, \mathit{Before}(\lambda\langle\rangle \,.\, x = 1 \wedge y = 1, \lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1)\rangle$

`endwhile`

$\langle 9, \mathit{Filter}(x \geq n, \lambda\langle\rangle \,.\, x = 0 \wedge y = 0 \sqcup$
$\mathit{Collect}(\lambda\langle\nu\rangle \,.\, x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0))\rangle$

Figure 15.3: Proving Conditional Correctness

15.3 A Program Reasoning Methodology

In this section, we introduce a program reasoning methodology based on the results of the previous section. The methodology attempts the proof of all program assertions, one by one, in an incremental manner, and is based on the following informal algorithm.

Let $\mathcal{K}$ be a verification problem

While $\mathcal{K}$ has unproven assertions

Let $\mathcal{K} := (\mathcal{T} \sqcap \mathcal{I})^k(\mathcal{K})$ for some $k \geq 0$

Pick a label $l \in \mathit{labels}(\mathcal{K})$ such that $\mathit{assert}_{\{l\}}(\cdots)$ appears at l

Pick a (possibly empty) set of labels $\Lambda \subseteq \mathit{labels}(\mathcal{K})$,

such that each $l' \in \Lambda$ appears in an *assert* construct

If $\mathit{assume}_{\Lambda \cup \{l\}}(\mathcal{T}(\mathcal{K}))|_l \vdash \mathit{assume}_l(\mathcal{K})|_l$ holds, then let $\mathcal{K} := \mathit{replace}(\mathcal{K}, l, \Lambda)$

Endwhile

We start with a verification problem $\mathcal{K}$. Throughout the reasoning process, the configuration $\mathcal{K}$ will change, on one hand due to the propagation steps applied to it, and on the other hand, due to the fact that the sets of labels appearing in $\mathit{assert}_\Lambda(\cdots)$ constructs will be changed to reflect only those labels whose assertions have not been proved yet. The while-loop condition in the informal algorithm states that, for as long as we have $\mathit{assert}_\Lambda(\cdots)$ constructs with non-empty sets of labels Λ in $\mathcal{K}$, the reasoning process may continue. Before attempting a to prove the conditional correctness, we may apply any number of propagation steps to the symbolic configuration. As argued in the previous chapter, this may improve the chances of achieving the proof. Then, we pick a program point l that has an unproven assertion, and a (possibly empty) set of labels Λ, and we attempt to prove that the assertion at l is conditionally correct on the assertions in Λ. If we succeed in doing this, we let the configuration reflect this fact by replacing the label l with the labels in Λ in all the *assert* constructs throughout the entire symbolic configuration $\mathcal{K}$. As a result, the set of labels appearing in *assert* constructs becomes smaller by one. When this set becomes empty, all assertions are proved *unconditionally*.

We illustrate this methodology with an example. Figure 15.1 shows a verification problem for a program that we have also analyzed in the previous chapter. This program manipulates the variables x and y in such a way that the parities of the two variables are synchronized. We regard this as a program fragment that shall be executed as part of a bigger program, and we are only interested in the executions which have x and y assigned to 0 at the beginning of the program fragment. To enforce this, we attach an assertion to the first program point stating the value of interest for variables x and y. We also attach an assertion to program point 8, stating the synchrony between the parities of x and y. First, we perform 4 steps of propagation on this symbolic configuration. The resulting configuration is shown in Figure 15.2. To make the configuration more readable, we have performed a few simplifications. First, we have replaced all the formulas of the form $\mathcal{F} \sqcap \lambda\langle\cdots\rangle\,.\,\mathit{true}$ by $\mathcal{F}$. Second, we have simplified all the *Assign* expressions. Let us denote by $\mathcal{K}_1$ the configuration in Figure 15.2. The next step is to compute $\mathcal{K}_2 = \mathit{assume}_{\{1,8\}}(\mathcal{T}(\mathcal{K}_1))$ and $\mathcal{K}_3 = \mathit{assume}_{\{8\}}(\mathcal{K}_1)$ and to check whether $K_2|_8 \vdash K_3|_8$ holds. If that happens, we have proved that the assertion at program point 8 is conditionally correct on the assertion at program point 1.

Figure 15.3 shows configuration $\mathcal{K}_2$, after some minor simplifications. Configuration K_3 is obtained by simply replacing every $\mathit{assert}_\Lambda(\mathcal{F})$ construct by $\mathcal{F}$. We now focus on program point 8 in both $\mathcal{K}_2$ and $\mathcal{K}_3$. We produce the proof obligation

$$\mathit{Before}(\lambda\langle\rangle\,.\,x = 1 \wedge y = 1, \lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1 \vdash \lambda\langle\nu\rangle\,.\,x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0.$$

This translates into the following first-order proof obligation

$$(\nu = 0 \rightarrow x = 1 \wedge y = 1) \wedge (\nu > 0 \rightarrow x \,\%\, 2 = 1 \leftrightarrow y \,\%\, 2 = 1) \rightarrow x \,\%\, 2 = 0 \leftrightarrow y \,\%\, 2 = 0,$$

which obviously holds.

Part IV

Finale

Chapter 16

Conclusion and Further Work

This chapter concludes the thesis. We start with an itemized summary of the entire work presented here, then we discuss future research in Section 16.2, and then we give a few concluding remarks in Section 16.3.

16.1 Summary

We started our work by recognizing several desirable features of a program reasoning framework. First, we wanted to be able to express a wide range of properties of programs, including but not limited to safety, liveness and behavioral properties. The distinction between safety and liveness properties has been defined by Lamport in [Lam77], who also argued that in reasoning about the correctness of sequential programs, safety properties and proof of termination are sufficient. However, as argued in Section 1.3, since we often need to reason about sequential programs that are part of larger, concurrent systems, being able to reason about liveness and behavioral properties of sequential programs is also important. Second, a program reasoning framework has to be compositional, since this allows a hierarchical decomposition of proofs in the case of large, complex systems. Third, since it is

not expected that a reasoning framework be fully automatic, the use of assertions should be permitted, as means of allowing the user to guide the proof along. Fourth, the framework should accommodate automated reasoning methods and allow their being combined with user-provided information. Finally, a reasoning framework should be incremental, in the sense that all the objectives should be achieved one by one, in separate phases, with the possibility of useful information being derived after every phase.

In order to achieve these features, we took the following steps:

- We devised the *configuration* data structure, as a versatile and compositional means of attaching information to program points. We then showed that the trace and progressive semantics can be expressed in terms of configurations. We also defined *progressive configurations* as configurations whose annotations are *indexed sets*, that is mappings from strings of natural numbers to sets of environments.

- We defined the *progressive semantics*, which is an abstraction of the trace semantics in the way that it projects sets of traces on program points, attaching to each program point a sequence of sets (i.e. an indexed set) of environments which define localized behavior; abstracting by projection achieves a degree of locality which contributes to the compositionality of the framework.

- We defined the progressive transfer function T^p, and showed that the progressive semantics of a program is the unique fixpoint of T^p.

- We introduced *family approximations*, as a technicality. If classic, superset-based approximations were used, the reasoning process would be too imprecise to allow the possibility of reaching the conclusion that a certain state *does indeed occur* (which is required when proving liveness and behavioral properties).

Families allow a more refined means of propagation, by which one could rule out that a program point is dead.

- Family approximations are also *partially ordered*; $\Phi_1 \preccurlyeq \Phi_2$ expresses the fact that the family Φ_1 is a more precise approximation than the family Φ_2.

- Next, we introduced *family configurations*, as means of approximating progressive configurations, and therefore, the progressive semantics of a program; we also lifted the $\preccurlyeq$ partial order to family configurations.

- We defined *family description languages* as means of specifying families. A family description language is accompanied by a formal system in which family description formulas can be reasoned about. We defined a set of properties that, if satisfied, would make a family description language useful in our framework. One important property is that, given two family description language formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, $\mathcal{F}_1 \models \mathcal{F}_2$ holds (i.e. can be proved) in the formal system whenever $[\![\mathcal{F}_1]\!] \preccurlyeq [\![\mathcal{F}_2]\!]$, where $[\![\mathcal{F}_1]\!]$ and $[\![\mathcal{F}_2]\!]$ are the families that represent the interpretations of the formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively.

- We then defined *symbolic configurations* as configurations whose annotations are family description formulas. The interpretation of a symbolic configuration is a family configuration, and a symbolic configuration is *correct* for a program whenever its interpretation is an approximation of the progressive semantics of the program.

- We then lifted the progressive operator to symbolic configurations. We showed that the symbolic propagation operator $\mathcal{T}$ can be the basis of a propagation calculus that would refine correct configurations, in the sense that, if the symbolic configuration $\mathcal{K}$ is correct of a given program, then $\mathcal{T}(\mathcal{K})$ is also

correct, and $\mathcal{T}(\mathcal{K}) \models \mathcal{K}$.

- To show that family description languages are useful, we took a side trip and showed that on top of a family description language we can define a *progressive Hoare calculus*.

- The next step was to introduce LAL (Liveness Aware Language) as an example of family description language. We showed that this language is powerful enough to express liveness properties of sequential programs through examples of propagation and Hoare-calculus proofs.

- Then, we introduced *assertion languages*. An assertion language can be defined on top of a family description language, by introducing the construct $assert_l(\mathcal{F})$, where l is a program point, and $\mathcal{F}$ is a family description formula. This construct expresses the fact that the user *believes* $\mathcal{F}$ to be true of the behavior of the program at l, while it has not proved it yet.

- In order to handle assertions, we introduce the concept of *conditional correctness*, and augment the propagation operator $\mathcal{T}$ such that it can handle symbolic configurations with assertions.

- One of the most important results in the thesis is Theorem 15.6, which states the condition under which an assertion can be derived from other assertions in the current symbolic configuration. This means that the assertion at hand is proved *conditionally*, under the assumption that other assertions that may not be proved yet, are in fact true. As far as the propagation process is concerned, the assertion at hand is redundant, and can be discarded. This result is formalized by Theorem 15.7.

- Theorem 15.7. leads to a *proof methodology* in which assertions are condition-

ally proved one by one, until no assertion is left. This realizes the incrementality feature that we mentioned in the first bullet of this itemized list.

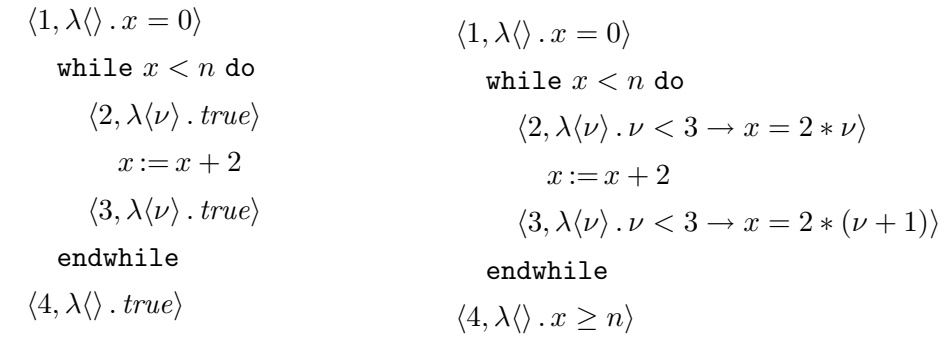
The result is a framework which uses symbolic configurations to express localized behavioral properties of programs. The propagation operator $\mathcal{T}$ can serve as means of refining configurations, and also conditionally proving assertions one by one, in an incremental manner. The framework is clearly compositional, as configurations are syntactic objects. Interestingly enough, during the propagation/refinement process, useful information can be derived after every step; we do not need the process to terminate (in fact it is not guaranteed to have that property) in order to obtain useful information about the program. Furthermore, as argued in the next section, the propagation operator could be abstracted, opening the door to realizing abstract interpretation within our framework.

16.2 Future Work

In this section we consider possible future developments of the ideas presented in this work. These fall into five major areas: integration of automated program reasoning methods, assertion refinement, AND the development of more expressive assertion languages. In what follows, we shall give a small description of each direction mentioned above, and a few hints at a possible solution.

Integrating Abstract Interpretation

We have argued throughout the thesis about the benefits of integrating automated program reasoning methods, like program analysis, into our framework. In this section we shall argue informally that abstract interpretation can be realized inside our framework by means of an abstract propagation operator. We start with a small



(a) Initial configuration (b) After 6 propagation steps

Figure 16.1: Propagation Produces Definite Information

example. Consider the configurations given in Figure 16.1. The initial configuration has a precondition stating that $x = 0$ at the beginning of the program, while the rest of the program points have the default formula annotation $\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, true$, where k is a suitably chosen arity. After applying 6 propagation steps, we get the configuration in Figure 16.1b. We note that in the progressive reasoning setting, propagation produces *definite* information. Indeed, for the first 3 rounds around the loop, the values of x at program point 2 will be 0,2, and 4. This is in contrast with program reasoning frameworks based on the collecting semantics, where definite information is "polluted" by useless information that is part of the greatest fixpoint of the propagation operator. However, the process of generating definite information in our framework is infinite. After 10 propagation steps, program point 2 shall have the formula annotation $\lambda\langle\rangle \,.\, \nu > 5 \rightarrow x = 2 * \nu$. While the upper limit of ν for which definite information is computed shall continually increase, the propagation process shall never compute the real invariant for program point 2, which is $\lambda\langle\nu\rangle \,.\, x = 2 * \nu$. This is where abstraction comes in. Instead of computing the sequence of values

$0, 2, 4, \ldots$ that are assigned to x, it may be more useful to infer, for example, that x is even throughout the execution of the program fragment. An abstract propagator would abstract the expression $2{*}\nu$ to *even* and that would lead to reaching a fixpoint in finite time.

Assertion Refinement

We have pointed out on several occasions that, in order to be able to prove what we want, we need configurations that are "refined" enough to allow that information to be inferred. To illustrate this situation, let us reprise the greatest common divisor example given in Section 1.3.

The configuration shown in Figure 1.2 was used as an example to show that our framework is able to prove termination. In order to come up with that configuration, we relied on a rather subtle observation about `if` statements. To propagate the star annotation through an `if` statement, the first order formula embedded in the annotation must either be implied by the `if` condition, or be implied by the negation of the `if` condition. In order to achieve annotations with this property, we added the extra parameter ν_2 to the configuration. The ν_2 parameter separates the environments occurring at program point 2 into those that have $a \leq b$ and those that have $a > b$. Thus, every slice of the indexed family at program point 2 represents an indexed set that goes entirely to program point 3 or entirely to program point 5 (that is, the slice is not split into two parts, one going to program point 3, and the other going to program point 5). Understanding the need for this extra parameter is not at all straightforward. So, whenever a proof of the greatest common divisor program would be attempted, the first configuration that comes to mind would be the one in Figure 16.2. In this configuration, the separation between environments that have $a \leq b$, and environments that have $a > b$ is missing, and as a result, the

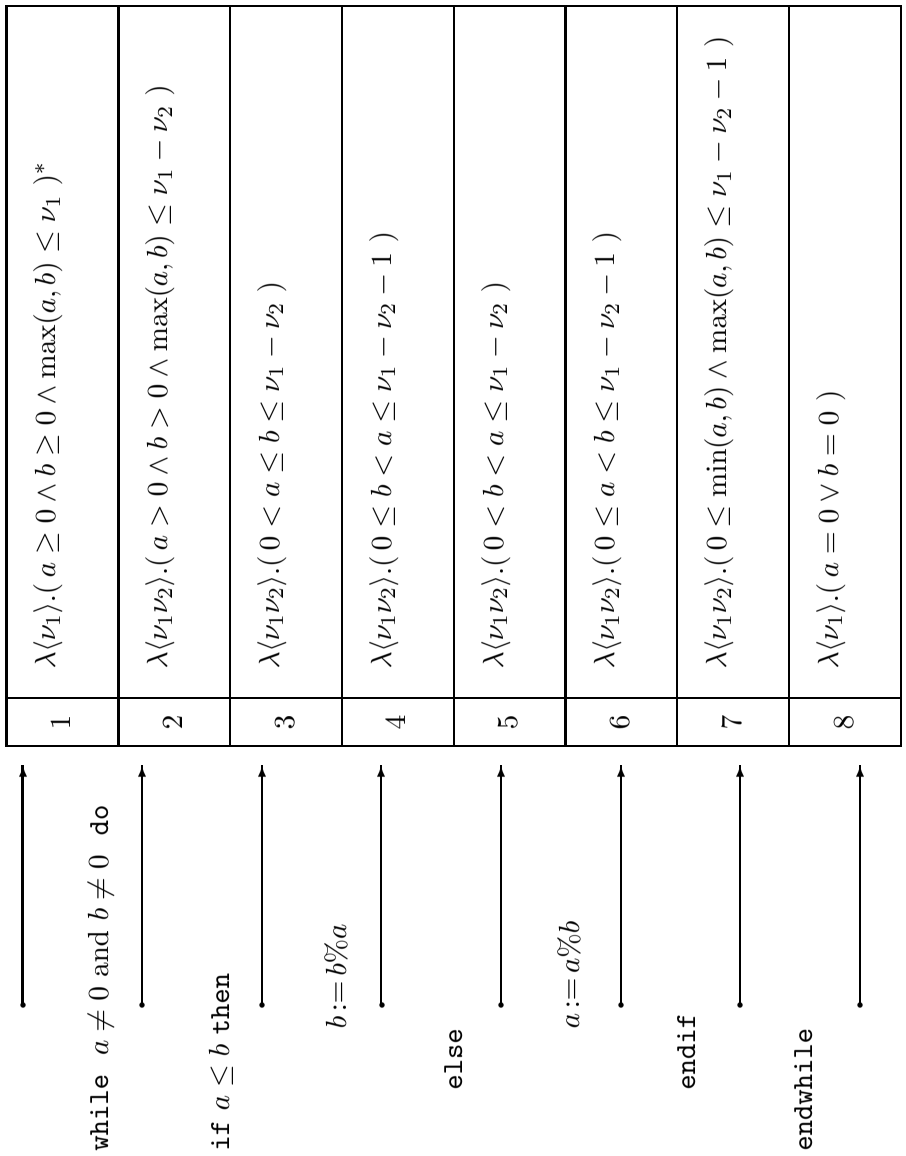


Figure 16.2: Assertion Refinement Example

$\langle 1, \lambda\langle\rangle . \quad \exists\forall\sigma . \exists\delta_1 . \forall\delta_2 . 0 \leq \delta_2 < 5 \rightarrow env(a)[\delta_1 + \delta_2] = 1 \wedge$

$\sigma(x) = 0 \wedge \sigma(z) \geq 0\rangle$

`while` $x < 100$ `do`

$\langle 2, \lambda\langle\rangle . \mathit{true}\rangle$

$z := (z + a[x]) \,\%\, 10$

$\langle 3, \lambda\langle\rangle . \mathit{true}\rangle$

$x := x + 1$

$\langle 4, \lambda\langle\rangle . \quad (\exists f . (\forall\delta_1 . \forall\delta_2 . \delta_1 \neq \delta_2 \rightarrow f(\delta_1) \neq f(\delta_2)) \wedge$

$\forall\delta . 0 \leq \delta < 5 \rightarrow \exists\sigma . \sigma(z) = f(\delta)) \wedge$

$\forall\sigma . 0 \leq \sigma(z) < 5\rangle$

`endwhile`

$\langle 5, \lambda\langle\rangle . \mathit{true}\rangle$

Figure 16.3: More Expressive Assertions

star annotation cannot be propagated through the `if` condition and termination cannot be inferred. However, once we have the configuration in Figure 16.2, it is possible to detect this propagation flaw, and attempt to correct it. This will result in the refined configuration given in Figure 1.2. One future development of this work would be to formalize the process of refining correct configurations such that the star annotation is not lost in the propagation process, and understand to what extent the process can be automated.

More Expressive Assertion Languages

In Chapter 11 we have argued that by using family configurations to approximate the progressive semantics of programs, we open the door to expressing properties that are much richer than liveness and safety. However, the ability to express such properties depends mainly on an expressive family description language. The

liveness aware language that we have introduced in Chapter 13, while expressive enough to represent liveness and safety in the same formula, are still far away from what families as means of approximation can do. An example of a more expressive language is given in Figure 16.3. The formula attached to program point 4 has several new elements, as compared to formulas in LAL. First, program variables are treated as constants (strings). The value of a program variable can be retrieved from an environment via an environment variable σ. Second, we note the fact that we have a higher order language, that allows functions to be quantified. These two features increase the expressive power of the family description language tremendously. Having environment variables allows us to state relationships between environments occurring at a given program point. For example, we could state that every time we reach a fixpoint, a variable is assigned a value that has not been assigned to it before, or that a variable has at least k distinct values throughout the execution of the program. In order to understand better how such properties are stated, let us take a closer look at the program in Figure 16.3. The program sums up modulo 10 the elements of an array into a variable z. The initial value of z is positive, and the array has at least 5 consecutive elements that have the value 1. The $\exists\forall$ quantifier specifies that the environment σ is unique. In other words, this program fragment is entered only once. The formula at program point 4 states that the variable z has at least 5 distinct values. That is, there exist at least 5 distinct environments whose values for the variable z are distinct. This is specified by stating that there exists some injective function f that maps the interval $[0, 4]$ into integers, and for all values $f(\delta)$, $0 \leq \delta < 5$, there exists an environment σ such that $\sigma(z) = f(\delta)$. Expressing such properties is often of interest in verifying embedded systems software, where programs do not typically terminate, and are required to comply with various resource access policies. For example, in a system with limited amount of

cache, the number of distinct values a variable has may be directly connected with the number of cache misses, so we either want to keep the number of distinct values small, so that it doesn't exceed the cache capacity, or we want to measure that number so as to determine the right size of the cache.

16.3 Concluding Remarks

To date, there is a wealth of techniques and methodologies for program reasoning, ranging from Hoare-style program verification, which is entirely user-driven, to the completely automatic, push-button program analysis and model-checking. Each of these techniques has certain useful features, and certain shortcomings. For instance, Hoare-style verification is compositional and allows proving the strongest properties of a program. On the other hand, this technique is entirely user-driven, and that makes it very difficult to apply Hoare-style verification to large programs. At the other end of the spectrum, program analysis methods are completely automatic. Most bottom-up program analysis methods, however, cannot derive correctness information about a program; they only derive certain general properties that may be useful in compiler optimizations, on in guiding various program transformations. Such method are also, in general, not incremental, that is, one has to wait until a program analysis procedure terminates in order to obtain any useful information.

When reasoning about large pieces of code, it is often the case that a very large portion of the code is concerned with error handling and the user interface, while a much smaller portion of the scale is concerned with implementing sophisticated algorithms. Reasoning about error handling and user interface code is often simple and can be handled by automated reasoning methods, while reasoning about sophisticated algorithms typically requires user assistance. The material presented in

this thesis has been centered around the belief that automated and non-automated reasoning techniques need to be combined within a general framework that would reduce the amount of user intervention and would allow reasoning about safety, liveness and temporal properties in a unified way.

The motivation for our work has been to create a framework that retains most of the desirable features of program reasoning techniques and methodologies in existence. Therefore, we have identified a set of desirable features of a program reasoning framework. The most important, in our view, is the ability to reason about safety, liveness, and temporal properties in a unified framework. Other important features are the ability to combine reasoning methods within one framework, and the reasoning framework being compositional and incremental. The main contributions of this thesis are:

- The *progressive semantics*, which is a programming language semantics which, as far as its level of abstraction is concerned, is hierarchically situated between the trace and collecting semantics. In our view, this semantics has the "right" level of abstraction, bringing together the compositionality of the collecting semantics and the ability to reason about liveness and (some abstraction of) temporal properties that are characteristic of the trace semantics.

- The concept of *family approximations*, as a more precise way of specifying the possible values of a set variable.

- The concept of *progressive reasoning* as a means of stating and verifying program properties, and a *compositional strongest postcondition propagation* calculus.

- A *liveness-aware* assertion language that is able to capture termination and

liveness properties, as well as some abstraction of the sequence of events that occur at a program point.

- The concept of *conditional correctness*, which allows a proof methodology that verifies assertions one by one, in an incremental manner.

Throughout our work, we have tried to keep a practical perspective. The results presented in this thesis are targeted at showing that it is possible to implement a program reasoning system that is modular and incremental, and open to integrating various program reasoning techniques. The *configuration* is the central data structure that we have used consistently throughout this work, and was devised with this purpose in mind. In our view, such a system applies (possibly abstract) propagation steps to a configuration that represents an approximation of the progressive semantics of a program which may contain assertions as means for a user to guide the reasoning along. By applying propagation steps, the configuration becomes more and more precise, and in this process, some of the assertions may become conditionally proved. Proving assertions may require the derivation of proof obligations which may be passed on to theorem provers like HOL and PVS. Automated program reasoning methods may be implemented as abstract propagation steps. One important property of such a system is that useful information may be derived from the configuration at hand after every propagation step.

This work has spawned several questions worth of being further investigated. We have sketched several of these problems in the previous section. However, keeping a practical perspective, we also need to extend the present framework to real programming languages and integrate the use of theorem provers.

Bibliography

[AAB+99] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems of combining abstraction and reachability analysis. In *Proc. 11th International Computer Aided Verification Conference*, pages 146–159, 1999.

[ABE00] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.

[ADG86] Krzysztof R Apt and C Delporte-Gallet. Syntax-Directed Analysis of Liveness Properties of While Programs. *Inform. and Control*, 68(1-3):223–253, January 1986.

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science*, 1145:52–??, 1996.

[Age92] S. Agerholm. Mechanizing program verification in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 208–222, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.

[Age94] O. Agesen. Constraint-based type inference and parametric polymorphism. *Lecture Notes in Computer Science*, 864:78–??, 1994.

[AHH93] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1993.

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming, Special Issue on SAS'96*, 35(1):79–111, September 1999.

[Ake78] S.B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, C-27(6), 1978.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986.

[AO97] Krzystof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag graduate texts in computer science series. Springer-Verlag, New York, NY, 2nd ed. edition, 1997.

[AP81] K. R. Apt and G. D. Plotkin. A Cook's tour of countable nondeterminzsm. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, volume 115 of *LNCS*, pages 479–494, Acre, Israel, July 1981. Springer.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt83] Krzysztof R. Apt. Formal justification of a proof system for communicating sequential processes. *Journal of the ACM*, 30(1):197–216, January 1983.

[BA98] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of POPL '98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–251. ACM SIGACT and SIGPLAN, ACM Press, 1998.

[BAMP83] M. Ben-Ari, Z. Manna, and Amir Pnueli. The temporal logic of branching time. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, volume 20, pages 207–226. ACM, ACM, 1983.

[BCC98] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, 1999.

[BCM95] M. Bruynooghe, M. Codish, and A. Mulkers. Abstracting unification: A key step in the design of logic program analyses. In *Computer Science Today*, volume 1000 of *LNCS*, pages 406–425. Springer-Verlag, 1995.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PwerPC microprocessor using symbolic model checking without BDDs. In *Proc. 11th International Computer Aided Verification Conference*, pages 60–71, 1999.

[BDM92] P. Bigot, S. K. Debray, and K. Marriott. Understanding Finiteness Analysis Using Abstract Interpretation. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 735–749, Washington, USA, November 1992. The MIT Press.

[BF97] D. Baldan and G. File. Abstract interpretation for improving WAM code. *Lecture Notes in Computer Science*, 1302:364–??, 1997.

[BFH91] A. Bouajjani, Jean-Claude Fernandez, and Nicholas Halbwachs. Minimal model generation. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 197–203, Berlin, Germany, June 1991. Springer.

[BFSV99] Roberto Barbuti, Nicoletta De Francesco, Antonella Santone, and Gigliola Vaglini. Formula based abstractions of transition systems for real-time model checking. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99—Formal Methods, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 289–306. Springer, 1999.

[BG99] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods in System Design: An International Journal*, 14(3):237–255, May 1999.

[BGL93] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *TOPLAS*, 16(5):133–181, January 1993.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proc. 9th International Computer Aided Verification Conference*, pages 400–411, 1997.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. *Lecture Notes in Computer Science*, 1302:172–??, 1997.

[Bir40] Garrett Birkhoff. Lattice theory. *Amer. Math. Soc. Coll. Pub.*, 25, 1940.

[Blu99] Matthias Blume. Dependency analysis for Standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4):790–812, November 1999.

[BM90] Robert S. Boyer and J. Strother Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 1–15, Kaiserslautern, FRG, July 1990. Springer Verlag.

[Bou90] Francois Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, volume 456 of *LNCS*, pages 307–323. Springer-Verlag, 1990.

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[Bou93a] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In Ian Sommerville and Manfred Paul, editors, *Software Engineering—ESEC '93*, volume 717 of *Lecture Notes in Computer Science*, pages 501–516. Springer-Verlag, 1993.

[Bou93b] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–??, 1993.

[Bou93c] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[BP00] James Bailey and Alexandra Poulovassilis. An abstract interpretation framework for termination analysis of active rules. *Lecture Notes in Computer Science*, 1949:252–??, 2000.

[Bra97] J. Brauburger. Automatic termination analysis for partial functions using polynomial orderings. *Lecture Notes in Computer Science*, 1302:330–??, 1997.

[Bur72a] R. M. Burstall. Program proving as hand simulation with a little induction. In *Proceedings IFIP Congress*, pages 308–312. North Holland, 1972.

[Bur72b] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CC77b] P Cousot and R Cousot. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, August 1977.

[CC77c] P Cousot and R Cousot. Static determination of dynamic properties of generalised type unions. In *Conference on Language Design for Reliable Software*, volume 12(3) of *ACM SIGPLAN Not.*, pages 77–94, March 1977.

[CC79a] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Math.*, 82(1):43–57, 1979.

[CC79b] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM, ACM, January 1979.

[CC92a] P Cousot and R Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP'92*, volume 631 of *LNCS*, pages 269–295. Springer-Verlag, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Abstract Interpretation and applications to logic programs. *J of Logic Programming*, 13(2-3):103–180, July 1992.

[CC92c] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC93a] P. Cousot and R. Cousot. "A la Burstall" intermittent assertions induction principles for proving inevitability properties of programs. *Theoretical Computer Science*, 120(1):123–155, 8 November 1993.

[CC93b] P. Cousot and R. Cousot. Galois connection-based abstract interpretations for strictness analysis, invited paper. In D. Bjørner, M. Broy, and I.V. Potossin, editors, *Proc. FMPA*, volume 735 of *LNCS*, pages 98–127, Academogorodok, Novosibirsk, RU, 28 June – 2 July 1993. Springer-Verlag.

[CC94] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional programs), invited paper. In *Proc. ICCL'94*, pages 95–112, Toulouse, FR, 16–19 May 1994. IEEE Computer Society Press.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7^{th} FPCA*, pages 170–181, La Jolla, CA, 25–28 June 1995. ACM Press.

[CC99] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, January 1999.

[CC00a] P. Cousot and R. Cousot. Abstract interpretation based program testing, invited paper. In Scuola Superiore G. Reis Romoli, editor, *Proc. SSGRR 2000 Computer and & eBusiness International Conference*, page compact disk paper 248, L'Aquila, IT, 31 Jul – 6 Aug 2000.

[CC00b] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, MA, January 2000. ACM Press.

[CCH94] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239. ACM SIGACT and SIGPLAN, ACM Press, 1994.

[CCMM95] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 75–84, 1995.

[CDG93a] M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *20th POPL*, pages 451–646, Charleston, SC, 1993. ACM Press.

[CDG93b] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 451–464. ACM SIGACT and SIGPLAN, ACM Press, 1993.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, volume 131 of *LNCS*. Springer-Verlag, Yorktown Heights, NY, May 1981.

[CF96] Robert Cartwright and Matthias Felleisen. Program verification through soft typing. *ACM Computing Surveys*, 28(2):349–351, June 1996.

[CFMW93] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. *Lecture Notes in Computer Science*, 700:633–??, 1993.

[CGLV99] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract interpretation based verification of logic programs. *ENTCS*, 30(1), 1999.

[CGLV03] Marco Comini, Roberta Gori, Giorgio Levi, and Paolo Volpe. Abstract interpretation based verification of logic programs. *Science of Computer Programming*, 49(1–3):89–123, December 2003.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CGS89] M. Codish, J. Gallagher, and E. Shapiro. Using safe approximations of fixed points for analysis of logic programs. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 233–262. MIT Press, 1989.

[CGS94] Thomas Cheatham, Haiming Gao, and Dan Stefanescu. A suite of analysis tools based on a general purpose abstract interpreter. *Lecture Notes in Computer Science*, 786:188–??, 1994.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, January 1978.

[Chu40] Alonzo Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CJM00] E. M. Clarke, S. Jha, and W. Marrero. Partial order reductions for security protocol verification. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000.

[CL96] Christopher Colby and Peter Lee. Trace-based program analysis. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 195–207. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[CM92] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report UCAM-CL-TR-265, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, August 1992.

[CMB+95] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria García de la Banda, and Manuel Hermenegildo. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[CMH91] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[Col96] Christopher P. Colby. *Semantics-based program analysis via symbolic composition of transfer relations /–Christopher Colby.* PhD thesis, Carnegie Mellon University, 1996.

[Con94] C. Consel. Fast strictness analysis via symbolic fixpoint iteration. In B. Le Charlerier, editor, *Proc. 1st Int. Symp. SAS '94*, volume 864 of *LNCS*, pages 423–431, Namur, BE, 20–22 Sep 1994. Springer-Verlag.

[Coo81] Stephen A. Cook. Corrigendum: "Soundness and completeness of an axiom system for program verification" [SIAM J. Comput. **7** (1978), no. 1, 70–90, MR **58** #13843]. *SIAM Journal on Computing*, 10(3):612–612, ???? 1981.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[Cou90] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[Cou97] Patrick Cousot. Types as abstract interpretations (invited paper). In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[Cou99] Patrick Cousot. Directions for research in approximate system analysis. *ACM Computing Surveys*, 31(3es):??–??, September 1999.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. *Lecture Notes in Computer Science*, 2000:138–??, 2001.

[Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[CRR+93] C. Zhang, R. Shaw, R. Olsson, K.N. Levitt, M. Archer, M. Heckman, and G. Benson. Mechanizing a programming logic for the concurrent programming language microSR in HOL. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 29–43, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[CS92] Thomas Cheatham and Dan Stefanescu. A suite of optimizers based on abstract interpretation. In *Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–81, San Francisco, U.S.A., June 1992. Association for Computing Machinery.

[CS98] M. Codish and H. Søndergaard. The boolean logic of set sharing analysis. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proc. 10^{th} Int. Symp. PLILP '98*, volume 1490 of *LNCS*, pages 89–101, Pisa, IT, 16–18 Sept. 1998. Springer-Verlag.

[Dah92] Ole-Johan Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice-Hall, New York, NY, 1992.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. 11th International Computer Aided Verification Conference*, pages 160–171, 1999.

[Deb94] S.K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, Int. Schools for Computer Scientists, pages 115–182. Clarendon Press, 1994.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48. ACM SIGACT and SIGPLAN, ACM Press, 1995.

[DGS96] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering*, pages 575–584. IEEE Computer Society Press / ACM Press, 1996.

[Dij72] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10), October 1972.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report #159, Compaq Systems Research Center, Palo Alto, USA, December 1998.

[DLSS00] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Automatic termination analysis of programs containing arithmetic predicates. *Electronic Notes in Computer Science vol. 30*, November 23 2000.

[DMW98] S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *25th POPL*, pages 12–24, San Diego, CA, 19–21 Jan 1998. ACM Press.

[DP97] Saumya K. Debray and Todd A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Transactions on Programming Languages and Systems*, 19(4):568–585, July 1997.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order: Second Edition*. Cambridge University Press, 2002.

[DS98] S. Decorte and D. De Schreye. Termination analysis: Some practical properties of the norm and level mapping space. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 235–249, Cambridge, June 15–19 1998. MIT Press.

[DW90] K. Davis and P. L. Wadler. Backwards strictness analysis: Proved and improved. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 12–30, London, UK, 1990. Springer-Verlag. British Computer Society Workshops in Computing Series.

[FA97] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In P. van Hentenryck, editor, *Proc. 4^{th} SAS'97*, volume 1302 of *LNCS*, pages 114–126, Paris, 8–10 Sep. 1997. Springer-Verlag.

[Fer00] Jérôme Feret. Confidentiality analysis of mobile systems. In Jens Palsberg, editor, *Proceedings of the 7th International Symposium on Static Analysis (SAS 2000), Santa Barbara, CA*, volume 1824 of *LNCS*, pages 135–154. Springer, 2000.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, 1997.

[FFK+96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Static debugging: Browsing the web of program invariants. In *Proc. PLDI'96*, pages 23–32, Philadelphia, PA, 21–24 May 1996. ACM Press.

[FH93] A. Ferguson and J. Hughes. Fast abstract interpretation using sequential algorithms. *Lecture Notes in Computer Science*, 724:45–??, 1993.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2–3):163–189, November 1999.

[FR99] G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Journal of Theoretical Computer Science*, 222(1-2):77–111, July 1999.

[FT89] Giunchiglia F. and Walsh T. Abstracting into inconsistent spaces (or the false proof problem). In Minnucci G., editor, *AI*IA 89*, Trento, 1989.

[GBP97] Richard Gerber, Tevfik Bultan, and William Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. In *CAV'97*, February 27 1997.

[GDL93] R. Giacobazzi, S. Debray, and G. Levi. Joining abstract and concrete computations in constraint logic programming. In M. Nivat, C. Rattray, and T. Rus, editors, *Proc. AMAST'93*, Workshops in Computing, pages 109–126, London, 1993. Springer-Verlag.

[GDL95] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–247, 1995.

[Gia96] Roberto Giacobazzi. "Optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *lncs*, pages 503–514, Grenoble, France, 22–24 February 1996. Springer.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. Research Report Spectre C - 40, LGI/IMAG, Grenoble, France, January 1993. Accepted for CAV'93.

[GM93] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gor88] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. Technical Report UCAM-CL-TR-145, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, September 1988.

[GR97] Roberto Giacobazzi and Francesco Ranzato. Making abstract interpretations complete. Technical Report TR-97-22, Dipartimento di Informatica, November 03 1997. Mon, 17 Nov 1997 10:20:30 GMT.

[Gra87] D. Gray. A pedagogical verification condition generator. *The Computer Journal*, 30(3):239–248, June 1987.

[Gra97] P. Granger. Static analyses of congruence properties on rational numbers. *Lecture Notes in Computer Science*, 1302:278–??, 1997.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Computer Aided Verification Conference*, pages 72–83, 1997.

[GV96] Fausto Giunchiglia and Adolfo Villafiorita. ABSFOL: a proof checker with abstraction. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 136–140, Berlin, July 30–August 3 1996. Springer.

[Han93] M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Tenth International Conference on Logic Programming*, pages 83–99. MIT Press, June 1993.

[Han95] M. Hanus. Compile-time analysis of nonlinear constraints in CLP($\mathcal{R}$). *New Generation Computing*, 13(2):155–186, 1995.

[Heh99] E.C.R. Hehner. Specifications, programs, and total correctness. *Science of Computer Programming*, 34:191–205, 1999.

[Hei92a] N. Heintze. Practical aspects of set-based analysis. In K.R. Apt, editor, *JICSLP*, pages 765–779, Washington, DC, November 1992. MIT Press.

[Hei92b] N. Heintze. *Set-Based Program Analysis*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, October 1992.

[Hei92c] Nevin Heintze. *Set-Based Program Analysis*. PhD thesis, Carnegie Mellon University, Pittsburgh , PA , USA, October 1992.

[Hei94] N. Heintze. Set-based analysis of ml programs. In *Proc. ACM Conf. on Lisp and Func. Prog.*, pages 306–317, Orlando, FL, 27–29 June 1994. ACM Press.

[Hei95] N. Heintze. Control-flow analysis and type systems. *Lecture Notes in Computer Science*, 983:189–206, 1995.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.

[Hen94] F. Henglein. Iterative fixed point computation for type-based strictness analysis. *Lecture Notes in Computer Science*, 864:395–??, 1994.

[Her94] M. Hermenegildo. Some methodological issues in the design of CIAO, a generic, parallel concurrent constraint logic programming system. *Lecture Notes in Computer Science*, 874:123–??, 1994.

[HJNN99] René Rydhof Hansen, Jacob Grydholt Jensen, Flemming Nielson, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. In Agostino Cortesi and Gilberto Filé, editors, *SAS'99: 6th International Static Analysis Sumposium (Venice, Italy)*, volume 1694 of *LNCS*, pages 134–148. Springer, 1999.

[HJV00] Nevin Heintze, Joxan Jaffar, and Răzvan Voicu. A framework for combining analysis and verification. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–39, Boston, Massachusetts, January 19–21, 2000.

[HL74a] 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.

[HL74b] C.A.R. Hoare and P.E. Lauer. Consistent and complimentary formal theories of the semantics of programming languages. *Acta Informatica*, 3(2):135–153, 1974.

[HL92] John Hughes and John Launchbury. Reversing Abstract Interpretations. In *ESOP'92*, volume 582 of *LNCS*, pages 269–286. Springer-Verlag, 1992.

[HM94] P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In *Proc. 7th International Higher Order Logic Theorem Proving and Its Applications Conference*, pages 269–284, 1994.

[HM98] P. V. Homeier and D. F. Martin. Mechanical verification of total correctness through diversion verification conditions. In *Proc. 11th International Theorem Proving in Higher Order Logics Conference*, pages 189–206, 1998.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real time-systems. In *Proc. 5^{th} LICS '92*. IEEE Comp. Soc. Press, June 1992.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, October 1969.

[Hoa78] C.A.R. Hoare. Some properties of predicate transformers. *Journal of the Association for Computing Machinery*, 25(3):461–480, July 1978.

[Hoa92] C. A. R. Hoare. Programs are predicates. In Institute for New Generation Computer Technology (ICOT), editor, *Proceedings of the International Conference on Fifth Generation Computer Systems. Volume 1*, pages 211–218, Burke, VA, USA, 1992. IOS Press.

[HP79] M. C. B. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *Lecture Notes in Computer Science 74*. Springer-Verlag, New York, NY, 1979.

[HPBLG02] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program debugging and validation using semantic approximations and partial specifications. *Lecture Notes in Computer Science*, 2380:69–??, 2002.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design: An International Journal*, 11(2):157–185, August 1997.

[HR98] Thomas A. Henzinger and Vlad Rusu. Reachability verification for hybrid automata. In Thomas A. Henzinger and Shankar Sastry, editors, *Hybrid Systems: Computation and Control (First International Workshop, HSCC'98)*, volume 1386, pages 190–204, Berkeley, CA, 1998. Springer-Verlag.

[HS95] Fritz Henglein and David Sands. A semantic model of binding times for safe partial evaluation. In S. D. Swierstra and M. Hermenegildo,

editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 299–320. Springer-Verlag, 1995.

[HS02a] Patricia M. Hill and Fausto Spoto. A foundation of escape analysis. *Lecture Notes in Computer Science*, 2422:380–??, 2002.

[HS02b] Patricia M. Hill and Fausto Spoto. A refinement of the escape property. *Lecture Notes in Computer Science*, 2294:154–??, 2002.

[HT98a] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine Subroutines. In Giorgio Levi, editor, *Static Analysis, 5th International Symposium, SAS'98*, number 1503 in Lecture Notes in Computer Science, pages 17–32. Springer-Verlag, Berlin Germany, 1998.

[HT98b] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. *Lecture Notes in Computer Science*, 1503:200–??, 1998.

[HTZ96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 208–219, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Hug92] S. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.

[HW72] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. Technical report, Eidgenossische Technische Hochschule, Zurich, Switzerland, November 1972.

[HY86] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 97–109. ACM, ACM, January 1986.

[ILL73] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification I: a logical basis and its implementation. Technical Report CS-TR-73-365, Stanford University, Department of Computer Science, May 1973.

[Jen95] Thomas P. Jensen. Clock analysis of synchronous dataflow programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 156–167, La Jolla, California, 21-23 June 1995.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[JM80] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *21st Annual Symposium on Foundations of Computer Science*, pages 185–190, Syracuse, New York, 13–15 October 1980. IEEE.

[Jon83] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon97] Neil D. Jones. Combining abstract interpretation and partial evaluation (brief overview). In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 396–405. Springer-Verlag, 1997.

[Kah87] G. Kahn. Natural semantics. In *Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435, 1997.

[KP99] Brian Kernighan and Rob Pike. Excerpt from The Practice of Programming: Finding performance improvements. *IEEE Software*, 16(2):61–65, March/April 1999.

[KP00] Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(4):328–342, 2000.

[Kri63] S. Kripke. A semantical analysis of modal logic i: normal modal propositional calculi. *Z. Math. Logik Grundladen Math.*, 9:67–96, 1963.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. *ACM SIGSOFT Software Engineering Notes*, 25(5):26–38, 2000.

[Lei01] K. Rustan M. Leino. Extended static checking: A ten-year perspective. *Lecture Notes in Computer Science*, 2000:157–??, 2001.

[Llo84] John Wylie Lloyd. *Foundataions of Logic Programming*. Springer-Verlag, Berlin, 1984.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for modula-3. In Kai Koskimies, editor, *Compiler Construction (CC'98)*, pages 302–305, Lisbon, 1998. Springer LNCS 1383.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.

[Mag93] Steve Maguire. *Writing solid code: Microsoft's techniques for developing bug-free C programs*. Microsoft Press, Bellevue, WA, USA, 1993.

[Mar90] Kim Marriott. Frameworks for Abstract Interpretation. Tech. Rep., IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[Mau00] Laurent Mauborgne. Tree schemata and fair termination. In J. Palsberg, editor, *Static Analyis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 302–320. Springer-Verlag, 2000.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[Mel85] C. Mellish. Some global optimizations for a prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[Mel94] Tom F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[Mos97] Christian Mossin. Exact flow analysis. In *Fourth International Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science (LNCS), pages 250–264, Paris, France, September 1997. Springer-Verlag.

[MOSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.

[MRB98] F. Malésieux, O. Ridoux, and P. Boizumault. Abstract compilation of λProlog. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 130–144, Cambridge, June 15–19 1998. MIT Press.

[MS93] K. Marriott and H. Søndergaard. On Propagation-Based Analysis of Logic Programs. In *ILPS Workshop on Global Compilation*, pages 45–65, Vancouver Canada, November 1993.

[MS94] K. Marriott and P.J. Stuckey. Approximating interaction between linear arithmetic constraints. In M. Bruynooghe, editor, *Proc. Int. Symp. ILPS 1994*, pages 571–585, Ithaca, NY, 13–17 Nov 1994. MIT Press.

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, September 1991.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need to call-by-value. In *Proceedings of the 4th International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer Verlag, 1980.

[Nau66] P. Naur. Proof of algorithms by general snapshots. *Nordisk tidskrift for informationsbehandling*, 6(4):310–316, 1966.

[Nec97a] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[Nec97b] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[Nie82] F. Nielson. A denotational framwork for data flow analysis. *Acta Informatica*, 18(3):265–287, 1982.

[Nie85] F. Nielson. Program transformations in a denotational setting. *ACM Transactions On Programming Languages And Systems*, 7(3):359–379, July 1985.

[Nie87] Flemming Nielson. Strictness analysis and denotational abstract interpretation. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 120–131, Munich, Germany, January 1987.

[NN88] H. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10(2):139–176, 1988.

[NN93] F. Nielson and H. R. Nielson. Finiteness conditions for strictness analysis. *Lecture Notes in Computer Science*, 724:194–??, 1993.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NS95] K. Nielsen and M. H. Sørensen. Call-by-name CPS-translation as a binding-time improvement. In A. Mycroft, editor, *International Static Analysis Symposium, Glasgow, Scotland, September 1995. (Lecture Notes in Computer Science, vol. 983)*. Berlin: Springer-Verlag, 1995.

[OCJ89] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[OSR95] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. CSL, 1995.

[Owi75] Susan S. Owicki. Axiomatic proof techniques for parallel programs. Technical Report TR75-251, Cornell University, Computer Science Department, July 1975.

[Pal93] J. Palsberg. Correctness of binding time analysis. *Journal of Functional Programming*, 3(3):347–363, July 1993.

[Pal95] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

[PDD97] S. Park, S. Das, and D.L. Dill. Automatic checking of aggregation abstractions through state enumeration. In *IFIP TC6/WG6.1 Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing, and Verification*, pages 207–222, November 1997.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Rag73] L. C. Ragland. *A verified program verifier*. PhD thesis, University of Texas at Austin, 1973.

[Rey98] J. C. Reynolds. *Theories of Programming Languages.* CUP, 1998.

[RK92] U. S. Reddy and S. N. Kamin. On the power of abstract interpretation. In *Proceedings: 4th International Conference on Computer Languages*, pages 24–33. IEEE Computer Society Press, 1992.

[RS99] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[Saï00] Hassen Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Seventh International Static Analysis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 377–396, Santa Barbara CA, June 2000. Springer-Verlag.

[Sch95] D. A. Schmidt. Natural-semantics-based abstract interpretation. *Lecture Notes in Computer Science*, 983:1–??, 1995.

[Sch98a] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, San Diego, California, January 19–21, 1998.

[Sch98b] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference Record of POPL '98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48. ACM SIGACT and SIGPLAN, ACM Press, 1998.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A Logic-Based Approach to Data Flow Analysis Problems. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 277–292. Springer, August 1990.

[Sha96] N. Shankar. Unifying verification paradigms. *Lecture Notes in Computer Science*, 1135:22–??, 1996.

[Søn86] H Søndergaard. An application of abstract interpretation of logic programs: occur-check reduction. In *ESOP'86, Saarbrucken, Germany*, volume 213 of *LNCS*, pages 327–338. Springer-Verlag, 1986.

[Spo01] Fausto Spoto. Watchpoint semantics: A tool for compositional and focussed static analyses. *Lecture Notes in Computer Science*, 2126:127–??, 2001.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 30 October 1996.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[SS99] H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc. 11th International Computer Aided Verification Conference*, pages 443–454, 1999.

[SSJ+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[Ste76] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math*, 5:285–309, 1955.

[Ten76] Robert D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.

[Tzo97] S. Tzolovski. Data dependences as abstract interpretation. *Lecture Notes in Computer Science*, 1302:366–??, 1997.

[VDS93] K. Verschaetse, S. Decorte, and D. De Schreye. Automatic termination analysis. In Kung-Kiu Lau and Tim Clement, editors, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing, pages 168–183, London, July 2–3 1993. Springer Verlag.

[Ven98] Arnaud Venet. Automatic determination of communication topologies in mobile systems. In Giorgio Levi, editor, *SAS'98: 5th International Static Analysis Symposium (Pisa, Italy)*, volume 1503 of *LNCS*, pages 152–167. Springer, 1998.

[Ven99] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2–3):223–248, November 1999.

[VS91] K. Verschaetse and D. De Schreye. Deriving Termination Proofs for Logic Programs, using Abstract Procedures. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 301–315, Paris, France, 1991. The MIT Press.

[VS99] Sofie Verbaeten and Danny De Schreye. Termination analysis of tabled logic programs using mode and type information. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS)*, volume 1722 of *Lecture Notes in Computer Science*, pages 163–178. Springer Verlag, 1999.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Moss, 1993.

[YcFG+00] And Y, J. c. Fernandez, L. Ghirvu, M. Bozga, and S. Bensalem. A transformational approach for generating non-linear invariants. In *SAS'2000*, December 14 2000.

[Zar97] F. Zartmann. Denotational abstract interpretation of functional logic programs. *Lecture Notes in Computer Science*, 1302:141–??, 1997.

Appendix A

Proofs in the thesis

5.2 Proposition Let P be a labeled program. The following statements hold.

a) $\langle \mathit{first}(P), \sigma \rangle$ is an initial state of P, for all environments $\sigma \in \Sigma$.

b) $\langle \mathit{last}(P), \sigma \rangle$ is a terminal state of P, for all environments $\sigma \in \Sigma$.

c) $(\mathbf{States}(P), \xrightarrow{P})$ is a deterministic transition system.

Proof: Statements a) and b) are proved by structural induction, considering all the five cases in Remark 4.1 In the base cases P is either a skip statement or an assignment. From the definition of $\xrightarrow{P}$, it results immediately that there is no state s of P such that $s \xrightarrow{P} \langle \mathit{first}(P), \sigma \rangle$ or $\langle \mathit{last}(P), \sigma \rangle \xrightarrow{P} s$, for all environments $\sigma \in \Sigma$.

If P a sequence program, then there exist programs P_1 and P_2 such that $P = P_1 \fatsemi P_2$, with $\mathit{last}(P_1) = \mathit{first}(P_2)$. From the induction hypothesis we have that statements a) and b) hold for both P_1 and P_2. Since $\mathit{first}(P) = \mathit{first}(P_1)$ and $\mathit{last}(P) = \mathit{last}(P_2)$, it follows that statements a) and b) hold for P as well.

If P is an `if` program, then there exist a constraint C, labels l_s, l_f and programs P_1, P_2 such that $P = \langle l_s \rangle$ `if` C `then` P_1 `else` P_2 `endif` $\langle l_f \rangle$. From the proper labeling

condition we have that $l_s, l_f \notin \mathit{labels}(P_1) \cup \mathit{labels}(P_2)$. Therefore, $\xrightarrow{P_1} \cup \xrightarrow{P_2}$ do not have transitions from and to states that have l_s or l_f as a label. In this case, the only transitions left are in $\xrightarrow{P\swarrow} \cup \xrightarrow{P\nearrow} \cup \xrightarrow{P\searrow} \cup \xrightarrow{P\nwarrow}$. None of these relations have transitions with destination states that have l_s as a label, or transitions with source states that have l_f as a label.

If P is a `while` program, then there exists a constraint C, labels l_s, l_f and a program P' such that $P = \langle l_s \rangle \,\texttt{while}\ C\ \texttt{do}\ P'\ \texttt{endwhile}\, \langle l_f \rangle$. From the proper labeling condition we have that $l_s, l_f \notin \mathit{labels}(P')$. The only transitions concerning labels l_s and l_f are in $\xrightarrow{P\searrow} \cup \xrightarrow{P\nwarrow} \cup \xrightarrow{P\circlearrowleft} \cup \xrightarrow{P\curvearrowright}$. None of these relations have transitions with destination states that have l_s as a label, or transitions with source states that have l_f as a label.

Statement c) shall be proved by structural induction as well. For the base cases, if P is a skip statement or an assignment, it is clear from the definition of $\xrightarrow{P}$ that for all environments σ, there exists exactly one state s such that $\langle \mathit{first}(P), \sigma \rangle \xrightarrow{P} s$. For the induction cases, assume first that P is a sequence program. Then, there exist programs P_1, P_2 such that $P = P_1 ; P_2$. Since the statement holds inductively for P_1 and P_2 and $\mathit{labels}(P) \backslash \{\mathit{last}(P)\} = (\mathit{labels}(P_1) \backslash \{\mathit{last}(P_1)\}) \cup (\mathit{labels}(P_2) \backslash \{\mathit{last}(P_2)\})$, it follows that the statement holds for P. Assume now that P is an `if` program, whose consequent and alternative are P_1 and P_2. We have that $\mathit{labels}(P) \backslash \{\mathit{last}(P)\} = (\mathit{labels}(P_1) \backslash \{\mathit{last}(P_1)\}) \cup (\mathit{labels}(P_2) \backslash \{\mathit{last}(P_2)\}) \cup \{\mathit{first}(P), \mathit{last}(P_1), \mathit{last}(P_2)\}$. The statement holds inductively for $\mathit{labels}(P_1) \backslash \{\mathit{last}(P_1)\}$ and $\mathit{labels}(P_2) \backslash \{\mathit{last}(P_2)\}$, and from the definitions of $\xrightarrow{P\swarrow}$, $\xrightarrow{P\searrow}$, $\xrightarrow{P\nearrow}$, and $\xrightarrow{P\nwarrow}$ it results that the statement holds for the labels $\mathit{first}(P)$, $\mathit{last}(P_1)$, and $\mathit{last}(P_2)\}$. Finally, assume that P is a `while` program, whose body is P'. We have that $\mathit{labels}(P) \backslash \{\mathit{last}(P)\} = (\mathit{labels}(P') \backslash \{\mathit{last}(P')\}) \cup \{\mathit{first}(P), \mathit{last}(P')\}$. The statement holds inductively for

labels in $labels(P')\backslash\{last(P')\}$ and from the definitions of $\xrightarrow{P\searrow}$, $\xrightarrow{P\nwarrow}$, $\xrightarrow{P\curvearrowright}$, and $\xrightarrow{P\circlearrowleft}$, the statement also holds for labels $first(P)$, and $last(P')$. □

7.1 Proposition Let $P = P_1 \,;\, P_2$ be a sequence program and $\theta \in \overrightarrow{P}$ be a trace. Exactly one of the following statements holds:

a) $\theta \sim \underbrace{t}_{P_1}$

b) $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{\langle last(P_1), \sigma \rangle t_2}_{P_2}$

Proof: We need to show that, on one hand, for every trace θ, either a) or b) are true, and, on the second hand, that a) and b) cannot be true simultaneously. The fact that a) and b) cannot be true simultaneously is clearly true, since a) states that θ is a trace of P_1, and therefore has no states that would belong to P_2, and on the other hand, b) implies that θ does have states belonging to P_2. Since θ cannot be in the situation where it would both have and not have states of P_2, it follows that a) and b) cannot be true simultaneously. The proof of the fact that the trace θ satisfies either a) or b) is by induction on the length of θ. For the base case, the length of θ is 1, which means that $\theta = \langle first(P), \sigma \rangle = \langle first(P_1), \sigma \rangle$, for some environment $\sigma \in \mathbf{Env}$. Clearly, a) is satisfied in this case. For the induction case, assume we have a trace $\theta \in \overrightarrow{P}$ of length $n > 1$, such that $\theta = \theta' s$, where θ' is the longest proper prefix of θ, and s is its last state. Clearly, the transition $\theta' \xrightarrow{P} s$ must be possible, so according to Proposition 5.2, the last state of θ' cannot have the label $last(P)$. From the induction hypothesis, we have that θ' satisfies either a) or b). Assume first that it satisfies a), i.e. θ' is a trace of P_1. Here we have two sub-cases: either the trace θ' ends at label $last(P_1)$, or it doesn't. If θ' ends at label $last(P_1)$, since $last(P_1) = first(P_2)$, according to Proposition 5.2, the transition $\theta' \xrightarrow{P_2} s$ must

be possible, and therefore θ satisfies b). For the second sub-case, assume that θ' is a trace of P_1, but it does not end at label $last(P_1)$. According to Proposition 5.2, the transition $\theta' \xrightarrow{P_1} s$ is possible, and therefore θ satisfies a). Assume now that θ' satisfies b), i.e. $\theta' \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{\langle last(P_1), \sigma \rangle t_2}_{P_2}$ holds. As mentioned above, θ' cannot end at label $last(P) = last(P_2)$. According to Proposition 5.2, the transition $\theta' \xrightarrow{P_2} s$ is possible, and therefore $\theta' \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{\langle last(P_1), \sigma \rangle t_2 s}_{P_2}$ holds. holds. It follows that θ satisfies b). $\square$

7.2 Proposition Let P be an `if` program whose condition is C and consequent and alternative are P_c and P_a. Consider a trace $\theta \in \overrightarrow{P}$. Exactly one of the following statements is true:

a) $\theta = \langle first(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$.

b) $\theta \sim \langle first(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models C$, and trace segment $t \in \overrightarrow{P_c}$.

c) $\theta \sim \langle first(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} \langle last(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models C$, and $t \in \overrightarrow{P_c}$.

d) $\theta \sim \langle first(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models \neg C$, and trace segment $t \in \overrightarrow{P_a}$.

e) $\theta \sim \langle first(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a} \xrightarrow{P\nwarrow} \langle last(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models \neg C$, and $t \in \overrightarrow{P_c}$.

Proof: A trace $\theta \in \overrightarrow{P}$ cannot satisfy simultaneously the five conditions given above. Condition a) requires that θ have exactly one state, while condition b) requires that θ have at least two states, with suffixes that are traces of P_c. Condition c) requires

that θ starts at label $\mathit{first}(P)$, ends at $\mathit{last}(P)$, and contains states of P_c. Conditions d) and e) are symmetric to b) and c) in the sense that they contain states of P_a, rather than P_c. It easy to see that no two conditions can be satisfied simultaneously by the same trace.

We are left with proving that every trace of P satisfies (at least) one of the conditions a) – e). This proof is by induction on the length of the trace. For the base case, assume we have a trace of length equal to 1, $\theta = \langle \mathit{first}(P), \sigma \rangle$, where $\sigma \in \mathbf{Env}$. Such a trace clearly satisfies a). For the induction case, assume that trace θ has a length greater than one. Denote by θ' the longest proper prefix of θ, that is $\theta = \theta' s$, where s is a state of P. From the induction hypothesis we have that θ' satisfies one of the conditions a) – e). In fact, because the transition $\theta' \xrightarrow{P} s$ must be possible, θ' cannot satisfy conditions c) or e). Assume θ' satisfies a), i.e. $\theta' = \langle \mathit{first}(P), \sigma \rangle$, for some environment $\sigma \in \mathbf{Env}$. Here we have two sub-cases, either $\sigma \models C$, or $\sigma \models \neg C$. If $\sigma \models C$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\swarrow} s$ holds, and therefore θ satisfies condition b). If $\sigma \models \neg C$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\searrow} s$ holds, and therefore θ satisfies condition d). Assume now that θ' satisfies condition b). Again, we have two sub-cases, either θ' ends at label $\mathit{last}(P_c)$ or it doesn't. If θ' ends at $\mathit{last}(P_c)$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} s$ holds, and it follows that θ satisfies condition c). If θ' does not end at $\mathit{last}(P_c)$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P_c} s$ holds, and it follows that θ satisfies condition b). The case when θ' satisfies condition d) has a similar proof with the case when θ' satisfies condition b). □

7.3 Proposition Let P be a `while` program whose condition is C and body is P_b. Consider a trace $\theta \in \vec{P}$. Exactly one of the following statements is true:

a) $\theta = \langle \mathit{first}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$.

b) $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\curvearrowright} \langle \mathit{last}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$, such that $\sigma \models \neg C$.

c) $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b}$, for some $k > 0$ and some environment $\sigma \in \mathbf{Env}$ such that $\sigma \models C$.

d) $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, for some $k > 0$ and some environments $\sigma_s, \sigma_f \in \mathbf{Env}$ such that $\sigma_s \models C$ and $\sigma_f \models \neg C$.

Proof: A trace $\theta \in \vec{P}$ cannot satisfy simultaneously the five conditions given above. Condition a) requires that θ have exactly one state, while condition b) requires that θ have two states, none of which be a state of P_b. Condition c) requires that θ ends with a state of P_b, while condition d) requires that θ ends at label $\mathit{last}(P)$ and contains states of P_b. It easy to see that no two conditions can be satisfied simultaneously by the same trace.

We are left with proving that every trace of P satisfies (at least) one of the conditions a) – e). This proof is by induction on the length of the trace. For the base case, assume we have a trace of length equal to 1, $\theta = \langle \mathit{first}(P), \sigma \rangle$, where $\sigma \in \mathbf{Env}$. Such a trace clearly satisfies a). For the induction case, assume that trace θ has a length greater than one. Denote by θ' the longest proper prefix of θ, that is $\theta = \theta' s$, where s is a state of P. From the induction hypothesis we have that θ' satisfies one of the conditions a) – d). In fact, because the transition $\theta' \xrightarrow{P} s$ must be possible, θ' cannot satisfy conditions b) and d). Assume θ' satisfies a), i.e. $\theta' = \langle \mathit{first}(P), \sigma \rangle$, for some environment $\sigma \in \mathbf{Env}$. Here we have two sub-cases, either $\sigma \models C$, or $\sigma \models \neg C$. If $\sigma \models C$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\searrow} s$ holds, and therefore θ satisfies condition c). If $\sigma \models \neg C$, then $\theta \sim \langle \mathit{first}(P), \sigma \rangle \xrightarrow{P\curvearrowright} s$ holds,

and therefore θ satisfies condition b). Assume now that θ' satisfies condition c). Again, we have two sub-cases, either θ' ends at label $last(P_b)$, or it doesn't. If θ' ends at label $last(P_b)$, from the definition of $\xrightarrow{P}$, the transition $\theta' \xrightarrow{P\nwarrow} s$ must be possible, and then it follows that θ satisfies condition d). If θ' does not end at label $last(P_b)$, then the transition $\theta' \xrightarrow{P_b} s$ must be possible, and it follows that θ satisfies condition c). □

7.7 Proposition Given a program P, let $\theta_1 = t_1\langle l, \sigma_1\rangle$ and $\theta_2 = t_2\langle l, \sigma_2\rangle$ be two traces of P ending at the same program point. Assume that θ_1 is a proper prefix of θ_2, and denote by $\tilde{\mu}_1$ and $\tilde{\mu}_2$ the progressive indices of θ_1 and θ_2, respectively. Then, $\tilde{\mu}_1 < \tilde{\mu}_2$.

Proof: The proof is by induction on the number of segments in the progressive segmentation of θ_2. For the base case, there is only one segment. Obviously, there will be only one segment in the progressive segmentation of θ_1 as well. Since θ_1 is a proper prefix of θ_2, and θ_2 ends with an occurrence of the label l, the number of occurrences of label l in θ_1 and θ_2 differ by at least 1. Therefore, $\tilde{\mu}_1 < \tilde{\mu}_2$.

For the inductive case, let $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle t_k\langle l_k, \sigma_k\rangle$ be the progressive segmentation of θ_2, and $t'_1\langle l'_1, \sigma'_1\rangle t'_2\langle l'_2, \sigma'_2\rangle \cdots t'_{j-1}\langle l'_{j-1}, \sigma'_{j-1}\rangle t'_j\langle l'_j, \sigma'_j\rangle$ be the progressive segmentation of θ_1. We can be in either of the following two situations:

a) $j = k$, $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle = t'_1\langle l'_1, \sigma'_1\rangle t'_2\langle l'_2, \sigma'_2\rangle \cdots t'_{j-1}\langle l'_{j-1}, \sigma'_{j-1}\rangle$, and $t'_j\langle l'_j, \sigma'_j\rangle$ is a prefix of $t_k\langle l_k, \sigma_k\rangle$;

b) $j < k$ and $t'_1\langle l'_1, \sigma'_1\rangle t'_2\langle l'_2, \sigma'_2\rangle \cdots t'_{j-1}\langle l'_{j-1}, \sigma'_{j-1}\rangle$ is a prefix of $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$.

In case a), let μ'_1 and μ'_2 be the last elements in the indices $\tilde{\mu}_1$ and $\tilde{\mu}_2$, respectively. By a line of reasoning similar to the one we used for the base case, we conclude $\mu'_1 < \mu'_2$, which entails $\tilde{\mu}_1 < \tilde{\mu}_2$

In case b), let $\tilde{\mu}'_1$ and $\tilde{\mu}'_2$ be the longest proper prefixes of $\tilde{\mu}_1$ and $\tilde{\mu}_2$. Clearly, $\tilde{\mu}'_1$ and $\tilde{\mu}'_2$ are the progressive indices of $t'_1\langle l'_1, \sigma'_1\rangle t'_2\langle l'_2, \sigma'_2\rangle \cdots t'_{j-1}\langle l'_{j-1}, \sigma'_{j-1}\rangle$ and $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_{k-1}\langle l_{k-1}, \sigma_{k-1}\rangle$, respectively. Using the induction hypothesis, $\tilde{\mu}'_1 < \tilde{\mu}'_2$, which again entails $\tilde{\mu}_1 < \tilde{\mu}_2$. □

7.8 Proposition The following three statements hold.

a) Given a sequence program $P = P_1 \,;\, P_2$ and a trace $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{t_2}_{P_2}$, the progressive index of θ w.r.t. P is the same as the progressive index of t_2 w.r.t. P_2.

b) Given an `if` program P, let P' be either its consequent or its alternative. Consider a trace $\theta \sim \langle \mathit{first}(P), \sigma\rangle \xrightarrow{P} \underbrace{t}_{P'}$. Then, the progressive index of θ w.r.t. P is the same as the progressive index of t w.r.t. P'.

c) Consider a `while` program P whose body is P'. Consider a trace $\theta \sim \langle \mathit{first}(P), \sigma\rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P'} \xrightarrow{P\circlearrowleft} \underbrace{t_2}_{P'} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P'}$, for some $k > 0$. Denote by $\tilde{\mu}$ the progressive index of t_k w.r.t. P'. Then, the progressive index of θ w.r.t. P is $(k-1)\tilde{\mu}$.

Proof: The proof of a) relies on the fact that t_1 starts at label $\mathit{first}(P_1)$ and ends at $\mathit{last}(P_1)$. Since $\mathit{first}(P_1) \ominus \mathit{last}(P_1)$, there is no label in P_1 that would be shallower than a label in P_2. Assume now that the progressive segmentation of t_2 is $t_{21}\langle l_1, \sigma_1\rangle t_{22}\langle l_2, \sigma_2\rangle \cdots t_{2k}\langle l_k, \sigma_k\rangle$. It is obvious that the progressive segmentation of θ is $t'_{21}\langle l_1, \sigma_1\rangle t_{22}\langle l_2, \sigma_2\rangle \cdots t_{2k}\langle l_k, \sigma_k\rangle$, where $t'_{21} = t_1 t_{21}$. The sequence of segments

$t_{22}\langle l_2, \sigma_2\rangle \cdots t_{2k}\langle l_k, \sigma_k\rangle$ appears in the progressive segmentations of both θ and t_2, and therefore they have the same progressive index.

The proof of b) proceeds in a similar manner. First we note that $\mathit{first}(P) \ni \mathit{first}(P')$. Therefore, if $t_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_k\langle l_k, \sigma_k\rangle$ is the progressive segmentation of the trace segment t w.r.t. P', then $t'_1\langle l_1, \sigma_1\rangle t_2\langle l_2, \sigma_2\rangle \cdots t_k\langle l_k, \sigma_k\rangle$ is the progressive segmentation of θ w.r.t. P, where t'_1 is $\langle \mathit{first}(P), \sigma\rangle t_1$. Since the segments $t_2\langle l_2, \sigma_2\rangle \cdots t_k\langle l_k, \sigma_k\rangle$ appear in the progressive segmentations of both θ and t, it follows that they have the same progressive index.

We provide only a proof outline for c) (a rigorous proof should be by induction on the number k of segments in the trace). First we note that $\mathit{first}(P) \circledcirc \mathit{first}(P')$. Assume that the progressive segmentation of t_k is $t_{k1}\langle l_1, \sigma_1\rangle t_{k2}\langle l_2, \sigma_2\rangle \cdots t_{km}\langle l_m, \sigma_m\rangle$, and denote its progressive index by $\tilde{\mu}$. The fact that $\mathit{first}(P) \circledcirc \mathit{first}(P')$ entails that $\mathit{first}(P) \circledcirc l_2$. Therefore, θ has the progressive segmentation $\epsilon\langle \mathit{first}(P), \sigma\rangle t'_{k1}\langle l_1, \sigma_1\rangle t_{k2}\langle l_2, \sigma_2\rangle \cdots t_{km}\langle l_m, \sigma_m\rangle$, where t'_{k1} is $t_1 \cdots t_{k-1}t_{k1}$. We note that the progressive segmentation of θ has an extra segment, perpended at the beginning of the trace, when compared to the progressive segmentation of t_k. This results in a progressive index that is longer by one position than $\tilde{\mu}$. The sequence of segments $k2\langle l_2, \sigma_2\rangle \cdots t_{km}\langle l_m, \sigma_m\rangle$ appears as a suffix in both progressive segmentations, and the $m-1$ segments in the suffix will produce the same numbers (i.e. the numbers appearing in $\tilde{\mu}$) in the progressive indices of θ and t_k. However, the extra segment $t'_{k1}\langle l_1, \sigma_1\rangle$ appearing in the progressive segmentation of θ will produce an extra number which has to be prepended to $\tilde{\mu}$. The value of that number is given by the number of progressive pairs in $t'_{k1} = t_1 \cdots t_{k-1}t_{k1}$. We note now that $t'_{k1} \sim \underbrace{t_1}_{P'} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_{k-1}}_{P'} \xrightarrow{P\circlearrowleft} \underbrace{t_{k1}}_{P'}$. We have a progressive pair every time a $\xrightarrow{P\circlearrowleft}$ transition is performed, which means that we have $k-1$ progressive pairs in

t'_{k1}. It follows that the progressive index of θ is $(k-1)\tilde{\mu}$. $\square$

7.9 Proposition Let $P = P_1 \fatsemi P_2$ be a sequence program and $\theta \in \vec{P}$ a trace. The following two statements are true.

a) If $\theta \sim \underbrace{t_1}_{P_1}$ and $rep_{P_1}(t_1) = \vec{K}_1$ for some singleton configuration $\vec{K}_1$ such that $|\vec{K}_1| = P_1$, then $rep_P(\theta) = \vec{K}_1 \fatsemi \vec{K}_{P_2,\perp}$.

b) If $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \langle last(P_1), \sigma\rangle \xrightarrow{P_2} \underbrace{t_2}_{P_2}$ such that $rep_{P_1}(t_1\langle last(P_1), \sigma\rangle) = \vec{K}_1$ and $rep_{P_2}(\langle last(P_1), \sigma\rangle t_2) = \vec{K}_2$ for some singleton configurations $\vec{K}_1$ and $\vec{K}_2$ with $|\vec{K}_1| = P_1$ and $|\vec{K}_2| = P_2$, then $rep_P(\theta) = \vec{K}_1 \fatsemi \vec{K}_2$.

Proof: In order to prove a) we need to show that the following two conditions hold:

(1) for every prefix θ' of θ ending in a state $\langle l, \sigma\rangle$ and whose progressive index is $\tilde{\mu}$, we have $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_l(\tilde{\mu}) = \{\sigma\}$;

(2) for all labels l' and progressive indices $\tilde{\mu}'$ for which there exists no prefix of θ ending at l' and whose progressive index would be $\tilde{\mu}'$, we have $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_{l'}(\tilde{\mu}') = \emptyset$.

First we note that every prefix of θ is a trace of both P and P_1, and has the same progressive index w.r.t both P and P_1. Consider a prefix θ' of θ ending with state $\langle l, \sigma\rangle$ and whose progressive index is $\tilde{\mu}$. Since $\vec{K}_1$ is the representation of θ w.r.t. P_1, we have that $\vec{K}_1|_l(\tilde{\mu}) = \{\sigma\}$. According to Remark 6.1, $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_l(\tilde{\mu}) = \vec{K}_1|_l(\tilde{\mu})$, which entails that $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_l(\tilde{\mu}) = \{\sigma\}$ and proves condition (1). In order to prove condition (2), we consider two cases: either $l' \in labels(P_1)$, or $l' \in labels(P_2)\backslash\{last(P_1)\}$. If $l' \in labels(P_1)$, since $rep_{P_1}(\theta) = \vec{K}_1$, it follows that $\vec{K}_1|_{l'}(\tilde{\mu}') = \emptyset$. Now, according to Remark 6.1, $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_{l'}(\tilde{\mu}') = \vec{K}_1|_{l'}(\tilde{\mu}')$, which

entails that $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_{l'}(\tilde{\mu}') = \emptyset$. Assume now that $l' \in \mathit{labels}(P_2)\backslash\{\mathit{last}(P_1)\}$. According to Remark 6.1, $(\vec{K}_1 \fatsemi \vec{K}_{P_2,\perp})|_{l'}(\tilde{\mu}') = \vec{K}_{P_2,\perp}|_{l'}(\tilde{\mu}') = \emptyset$.

In order to prove b), we need to show that the following two conditions hold:

(1') for every prefix θ' of θ ending in a state $\langle l, \sigma\rangle$ and whose progressive index is $\tilde{\mu}$, we have $(\vec{K}_1 \fatsemi \vec{K}_2)|_l(\tilde{\mu}) = \{\sigma\}$;

(2') for all labels l' and progressive indices $\tilde{\mu}'$ for which there exists no prefix of θ ending at l' and whose progressive index would be $\tilde{\mu}'$, we have $(\vec{K}_1 \fatsemi \vec{K}_2)|_{l'}(\tilde{\mu}') = \emptyset$.

We consider two cases: either $l, l' \in \mathit{labels}(P_1)$, or $l, l' \in \mathit{labels}(P_2)\backslash\{\mathit{last}(P_1)\}$. The proof for $l, l' \in \mathit{labels}(P_1)$ is similar to the one for condition a) and will be omitted. We first prove condition (1') for $l, l' \in \mathit{labels}(P_2)\backslash\{\mathit{last}(P_1)\}$. In this case, $\theta' \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \langle \mathit{last}(P_1), \sigma\rangle \xrightarrow{P_2} \underbrace{t_2'}_{P_2}$. According to Proposition 7.8, the progressive indices of θ w.r.t. P and t_2' w.r.t. P_2 are the same, and therefore equal to $\tilde{\mu}$. Since $\langle \mathit{last}(P_1), \sigma\rangle t_2'$ is a prefix of $\langle \mathit{last}(P_1), \sigma\rangle t_2$ and $\mathit{rep}_{P_2}(\langle \mathit{last}(P_1), \sigma\rangle t_2) = \vec{K}_2$, it follows that $\vec{K}_2|_l(\tilde{\mu}) = \{\sigma\}$. Now, since $l \in \mathit{labels}(P_2)\backslash\{\mathit{last}(P_1)\}$, according to Remark 6.1, $(\vec{K}_1 \fatsemi \vec{K}_2)|_l(\tilde{\mu}) = \vec{K}_2|_l(\tilde{\mu}) = \{\sigma\}$, which proves condition (1'). Condition (2') follows from the fact that, since there exists no prefix of θ ending at l' and with progressive index $\tilde{\mu}$, there exists no prefix of $\langle \mathit{last}(P_1), \sigma\rangle t_2$ ending at l' and with progressive index $\tilde{\mu}$. This entails that $\vec{K}_2|_l(\tilde{\mu}) = \emptyset$ which, according to Remark 6.1, leads to $(\vec{K}_1 \fatsemi \vec{K}_2)|_l(\tilde{\mu}) = \emptyset$. $\square$

7.10 Proposition Let P be an `if` program whose condition is C and consequent and alternative are P_c and P_a, respectively. Given a trace $\theta \in \vec{P}$, the following statements are true.

a) If $\theta = \langle \mathit{first}(P), \sigma \rangle$ for some environment $\sigma \in \mathbf{Env}$, then $\mathit{rep}_P(\theta) =$
$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma\} \rangle \,\mathtt{if}\ C\ \mathtt{then}\ \vec{K}_{P_c,\perp}\ \mathtt{else}\ \vec{K}_{P_a,\perp}\ \mathtt{endif}\, \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset \rangle$.

b) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models C$, and trace segment $t \in \vec{P_c}$, and if $\mathit{rep}_{P_c}(t) = \vec{K}_c$, where $\vec{K}_c$ is a configuration such that $|\vec{K}_c| = P_c$, then $\mathit{rep}_P(\theta) =$
$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \,\mathtt{if}\ C\ \mathtt{then}\ \vec{K}_c\ \mathtt{else}\ \vec{K}_{P_a,\perp}\ \mathtt{endif}\, \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset \rangle$.

c) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} \langle \mathit{last}(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models C$, and $t \in \vec{P_c}$, and if $\mathit{rep}_{P_c}(t) = \vec{K}_c$, where $\vec{K}_c$ is a configuration such that $|\vec{K}_c| = P_c$, then $\mathit{rep}_P(\theta) =$
$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \,\mathtt{if}\ C\ \mathtt{then}\ \vec{K}_c\ \mathtt{else}\ \vec{K}_{P_a,\perp}\ \mathtt{endif}\, \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\} \rangle$.

d) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a}$, for some environment $\sigma_s \in \mathbf{Env}$ such that $\sigma_s \models \neg C$, and trace segment $t \in \vec{P_a}$, and if $\mathit{rep}_{P_a}(t) = \vec{K}_a$, where $\vec{K}_a$ is a configuration such that $|\vec{K}_a| = P_a$, then $\mathit{rep}_P(\theta) =$
$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \,\mathtt{if}\ C\ \mathtt{then}\ \vec{K}_{P_c,\perp}\ \mathtt{else}\ \vec{K}_a\ \mathtt{endif}\, \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset \rangle$.

e) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, where $\sigma_s, \sigma_f \in \mathbf{Env}$, with $\sigma_s \models \neg C$, and $t \in \vec{P_a}$, and if $\mathit{rep}_{P_a}(t) = \vec{K}_a$, where $\vec{K}_a$ is a configuration such that $|\vec{K}_a| = P_a$, then
$\mathit{rep}_P(\theta) \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\} \rangle \,\mathtt{if}\ C\ \mathtt{then}\ \vec{K}_{P_c,\perp}\ \mathtt{else}\ \vec{K}_a\ \mathtt{endif}\, \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\} \rangle$.

Proof: The proof of a) is trivial, since for case the trace θ has only the state $\langle \mathit{first}(P), \sigma \rangle$ whose progressive index is ϵ. Therefore, the configuration representing the trace θ will have the indexed set $\lambda\langle\rangle \,.\, \{\sigma\}$ as the annotation associated with the label $\mathit{first}(P)$, and the empty indexed set for all the other labels. In what follows, we shall present only the proofs for conditions b) and c). The proofs for conditions d) and e) are similar to the proofs of conditions b) and c) and shall be omitted.

Both conditions b) and c) claim that a certain trace θ is represented by a specific singleton configuration $\vec{K}$. In order to prove such a claim, we need to show that the following two conditions hold.

(1) for every prefix θ' of θ ending in a state $\langle l, \sigma\rangle$ and whose progressive index is $\tilde{\mu}$, we have $\vec{K}|_l(\tilde{\mu}) = \{\sigma\}$;

(2) for all labels l' and progressive indices $\tilde{\mu}'$ for which there exists no prefix of θ ending at l' and whose progressive index would be $\tilde{\mu}'$, we have $\vec{K}|_{l'}(\tilde{\mu}') = \emptyset$.

We proceed now with the proof of condition b). For convenience, let us denote by $\vec{K}$ the configuration $\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_c$ `else` $\vec{K}_{P_a,\perp}$ `endif` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset\rangle$. Let θ' be a prefix of θ. Here we have two sub-cases. Either $\theta' = \langle \mathit{first}(P), \sigma_s\rangle$, or $\theta' \sim \langle \mathit{first}(P), \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t'}_{P_c}$, where t' is a prefix of t. If $\theta' = \langle \mathit{first}(P), \sigma_s\rangle$, then its progressive index is ϵ, and it is easy to verify that $\vec{K}|_{\mathit{first}(P)}(\epsilon) = \{\sigma_s\}$. If $\theta' \sim \langle \mathit{first}(P), \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t'}_{P_c}$, where t' is a prefix of t, then according to Proposition 7.8, the progressive index of θ' is equal to the progressive index of t'. Denote this progressive index by $\tilde{\mu}$, and denote by $\langle l, \sigma\rangle$ the last state of both θ' and t'. Since $\vec{K}_c$ is the representation of t, it must be the case that $\vec{K}_c|_l(\tilde{\mu}) = \{\sigma\}$. Remark 7.8 entails that $\vec{K}|_l = \vec{K}_c|_l$, and therefore we have $\vec{K}|_l = \{\sigma\}$. Thus, condition (1) is satisfied. We are left with proving that for all labels $l' \in \mathit{labels}(P)$, or indices $\tilde{\mu}'$, such that there is no prefix of θ that would end at label l', or have progressive index $\tilde{\mu}'$, we have $\vec{K}|_{l'}(\tilde{\mu}') = \emptyset$. Here we have four sub-cases: either $l' = \mathit{first}(P)$, or $l' \in \mathit{labels}(P_c)$, or $l' \in \mathit{labels}(P_a)$, or $l' = \mathit{last}(P)$. The only interesting sub-case is when $l' \in \mathit{labels}(P_c)$; all the other sub-cases are trivial. If $l' \in \mathit{labels}(P_c)$, but there is no prefix of θ ending at l', or whose progressive index is $\tilde{\mu}'$, it follows that there is no prefix of t ending at l', or whose progressive index

is $\tilde{\mu}'$. Since $\vec{K}_c$ is the representation of t, it must be the case that $\vec{K}_c|_l(\tilde{\mu}) = \emptyset$. Remark 7.8 entails that $\vec{K}|_l = \vec{K}_c|_l$, and therefore we have $\vec{K}|_l = \emptyset$. This completes the proof of condition b).

For the proof of condition c) we will again conveniently denote by $\vec{K}$ the configuration

$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `if` C `then` $\vec{K}_c$ `else` $\vec{K}_{P_a,\perp}$ `endif` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\}\rangle$. Let θ' be a prefix of θ. First we prove that condition (1) given above holds. Here we have three sub-cases. Either $\theta' = \langle \mathit{first}(P), \sigma_s\rangle$, or $\theta' \sim \langle \mathit{first}(P), \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t'}_{P_c}$, where t' is a prefix of t, or $\theta' = \theta$. The proofs of the first two sub-cases are identical to the corresponding sub-cases in the proof of condition b). For the sub-case when $\theta' = \theta$, the progressive index of θ is ϵ, and it is immediate to verify that $\vec{K}|_{\mathit{last}(P)}(\epsilon) = \{\sigma_s\}$. In order to show that condition (2) holds, we have four cases, identical to the ones in the proof of condition b). $\square$

7.11 Proposition Let P be a `while` program whose condition and body are C and P_b, respectively. Given a trace $\theta \in \vec{P}$, the following statements are true.

a) If $\theta = \langle \mathit{first}(P), \sigma\rangle$ for some environment $\sigma \in$ **Env** then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma\}\rangle$ `while` C `do` $\vec{K}_{P_b,\perp}$ `endwhile` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset\rangle$.

b) If $\theta \sim \langle \mathit{first}(P), \sigma\rangle \xrightarrow{P\curvearrowright} \langle \mathit{last}(P), \sigma\rangle$ for some environment $\sigma \in$ **Env**, such that $\sigma \models \neg C$ then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma\}\rangle$ `while` C `do` $\vec{K}_{P_b,\perp}$ `endwhile` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma\}\rangle$.

c) If $\theta \sim \langle \mathit{first}(P), \sigma_s\rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b}$, for some $k \geq 0$ and some environment $\sigma \in$ **Env** such that $\sigma \models C$, and if $\mathit{rep}_{P_b}(t_i) = \vec{K}_i$, $1 \leq i \leq k$, where $\vec{K}_i$ is a configuration such that $|\vec{K}_i| = P_c$, then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `while` C `do` $\mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k)$ `endwhile` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset\rangle$.

d) If $\theta \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle \mathit{last}(P), \sigma_f \rangle$, for some $k \geq 0$ and some environments $\sigma_s, \sigma_f \in \mathbf{Env}$ such that $\sigma_s \models C$ and $\sigma_f \models \neg C$, and if $\mathit{rep}_{P_b}(t_i) = \vec{K}_i$, $1 \leq i \leq k$, where $\vec{K}_i$ is a configuration such that $|\vec{K}_i| = P_c$, then $\mathit{rep}_P(\theta) = \langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `while` C `do` $\mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k)$ `endwhile` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\}\rangle$.

Proof: We have to prove that a trace θ is represented by a specific singleton configuration $\vec{K}$. In order to prove such a claim, we need to show that the following two conditions hold.

(1) for every prefix θ' of θ ending in a state $\langle l, \sigma \rangle$ and whose progressive index is $\tilde{\mu}$, we have $\vec{K}|_l(\tilde{\mu}) = \{\sigma\}$;

(2) for all labels l' and progressive indices $\tilde{\mu}'$ for which there exists no prefix of θ ending at l' and whose progressive index would be $\tilde{\mu}'$, we have $\vec{K}|_{l'}(\tilde{\mu}') = \emptyset$.

Conditions a) and b) assume that the trace θ has one state, and two states, respectively. Verifying that the two conditions hold is similar with verifying that condition a) in Proposition 7.10 holds. We shall focus on proving conditions c) and d).

We proceed now with the proof of condition c). For convenience, let us denote by $\vec{K}$ the configuration

$$\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle \texttt{ while } C \texttt{ do } \mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k) \texttt{ endwhile } \langle \mathit{last}(P), \lambda\langle\rangle \,.\, \emptyset\rangle.$$

Let θ' be a prefix of θ. Here we have two sub-cases. Either $\theta' = \langle \mathit{first}(P), \sigma_s \rangle$, or $\theta' \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t'_i}_{P_b}$, where $i \leq k$ and t'_i is a prefix of t_i. If $\theta' = \langle \mathit{first}(P), \sigma_s \rangle$, then its progressive index is ϵ, and it is easy to verify that $\vec{K}|_{\mathit{first}(P)}(\epsilon) = \{\sigma_s\}$. Assume now that $\theta' \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow}$

$\underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t'_i}_{P_b}$, where $i \leq k$ and t'_i is a prefix of t_i. Denote by $\tilde{\mu}$ the progressive index of θ' w.r.t. P, and by $\tilde{\mu}'$ the progressive index of t'_i w.r.t. P_b. According to Proposition 7.8, we have $\tilde{\mu} = i\tilde{\mu}'$. Denote by $\langle l, \sigma \rangle$ the state at the end of both θ' and t'_i. Since $\vec{K}_i$ is the representation of t_i w.r.t. P_b, it follows that $K_i|_l(\tilde{\mu}') = \{\sigma\}$. According to Remark 6.1, $\vec{K}|_l(\tilde{\mu}) = \mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k)|_l(\tilde{\mu}) = \vec{K}_i|_l(\tilde{\mu}') = \{\sigma\}$. Thus, condition (1) is satisfied. We are left with proving that for all labels $l' \in \mathit{labels}(P)$, or indices $\tilde{\mu}'$, such that there is no prefix of θ that would end at label l', or have progressive index $\tilde{\mu}'$, we have $\vec{K}|_{l'}(\tilde{\mu}') = \emptyset$. We now have three cases: either $l' = \mathit{first}(P)$, or $l' \in \mathit{labels}(P_b)$, or $l' = \mathit{last}(P)$. The cases when $l' = \mathit{first}(P)$ or $l' = \mathit{last}(P)$ are trivial, and we shall focus only on the case when $l' \in \mathit{labels}(P_b)$. Assume, by way of contradiction, that $\vec{K}|_{l'}(\tilde{\mu}') = \{\sigma'\}$. Since l' is in $\mathit{labels}(P_b)$ and $\mathit{first}(P) \circledcirc \mathit{first}(P_b)$, it follows that $\tilde{\mu}' \neq \epsilon$. Assume that $\tilde{\mu}' = i\tilde{\mu}''$, where $i \geq 0$. Then, by Proposition 7.8, $\vec{K}_i|_{l'}(\tilde{\mu}'') = \{\sigma\}$. Now, $\vec{K}_i$ is the representation of t_i, and this entails that there exists a prefix of t_i, call it t'_i, which ends with the state $\langle l', \sigma' \rangle$, and whose progressive index is $\tilde{\mu}''$. Then, $\langle \mathit{first}(P), \sigma_s \rangle t_0 t_1 \cdots t_{i-1} t'_i$ is a prefix of θ that ends with the state $\langle l', \sigma' \rangle$, and whose progressive index is $\tilde{\mu}'$, and this leads to a contradiction. It follows that condition (2) holds.

For the proof of condition d) we will conveniently denote by $\vec{K}$ the configuration $\langle \mathit{first}(P), \lambda\langle\rangle \,.\, \{\sigma_s\}\rangle$ `while` C `do` $\mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k)$ `endwhile` $\langle \mathit{last}(P), \lambda\langle\rangle \,.\, \{\sigma_f\}\rangle$. Let θ' be a prefix of θ. First we prove that condition (1) given above holds. Here we have three sub-cases. Either $\theta' = \langle \mathit{first}(P), \sigma_s \rangle$, or $\theta' \sim \langle \mathit{first}(P), \sigma_s \rangle \xrightarrow{P\searrow} \underbrace{t_0}_{P_b} \xrightarrow{P\circlearrowleft} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t'_i}_{P_b}$, where $i \leq k$ and t'_i is a prefix of t_i, or $\theta' = \theta$. The proofs for the first two cases are identical to the corresponding cases of condition c). For the sub-case when $\theta' = \theta$, the progressive index of θ is ϵ, and it is immediate to verify that $\vec{K}|_{\mathit{last}(P)}(\epsilon) = \{\sigma_s\}$. In order to show that condition (2) holds, we have

three cases, identical to the ones in the proof of condition c). □

7.16 Proposition Let P be a program and $\vec{K}_1, \ldots, \vec{K}_k$ and $\vec{K}'_1, \ldots, \vec{K}'_k$ two sets of singleton configurations such that $|\vec{K}_i| = |\vec{K}'_i| = P$, for all i, $1 \leq i \leq k$. Assume that $\vec{K}_i \longrightarrow \vec{K}'_i$. Then, $seq(\vec{K}_1, \ldots, \vec{K}_i) \longrightarrow seq(\vec{K}'_1, \ldots, \vec{K}'_i)$, $k > 0$.

Proof: The proof is by structural induction on the program P. For convenience, denote $seq(\vec{K}_1, \ldots, \vec{K}_i)$ and $seq(\vec{K}'_1, \ldots, \vec{K}'_i)$, by $\vec{K}$ and $\vec{K}'$, respectively. Assume first that P is the skip program $\langle l_s \rangle \, \texttt{skip} \, \langle l_f \rangle$. Then, we can write $\vec{K}_i$ and $\vec{K}'_i$ as $\langle l_s, \Psi_{si} \rangle \, \texttt{skip} \, \langle l_f, \Psi_{fi} \rangle$ and $\langle l_s, \Psi_{si} \rangle \, \texttt{skip} \, \langle l_f, \Psi_{si} \rangle$, respectively, for all i, $1 \leq i \leq k$. According to Remarks 7.14 and 7.15, we have that

$$seq(\vec{K}_1, \ldots, \vec{K}_i) = \langle l_s, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle \, \texttt{skip} \, \langle l_f, seq(\Psi_{f1}, \ldots, \Psi_{fk}) \rangle$$

and

$$seq(\vec{K}'_1, \ldots, \vec{K}'_i) = \langle l_s, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle \, \texttt{skip} \, \langle l_f, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle,$$

from which it follows that $\vec{K} \longrightarrow \vec{K}'$. Next, assume that P is the assignment $\langle l_s \rangle \, x := E \, \langle l_f \rangle$. Then, we can write $\vec{K}_i$ and $\vec{K}'_i$ as $\langle l_s, \Psi_{si} \rangle \, x := E \, \langle l_f, \Psi_{fi} \rangle$ and $\langle l_s, \Psi_{si} \rangle \, x := E \, \langle l_f, assign(x, E, \Psi_{si}) \rangle$, respectively, for all i, $1 \leq i \leq k$. According to Remarks 7.14 and 7.15 we have that

$$seq(\vec{K}_1, \ldots, \vec{K}_k) = \langle l_s, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle \, x := E \, \langle l_f, seq(\Psi_{f1}, \ldots, \Psi_{fk}) \rangle$$

and

$$seq(\vec{K}'_1, \ldots, \vec{K}'_i) = \langle l_s, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle \, x := E \, \langle l_f, assign(x, E, seq(\Psi_{s1}, \ldots, \Psi_{sk})) \rangle,$$

from which it follows that $\vec{K} \longrightarrow \vec{K}'$. Assume now that P is the sequence program $P_1 \fatsemi P_2$. Then, we can write $\vec{K}_i$ and $\vec{K}'_i$ as $\vec{K}_{i1} \fatsemi \vec{K}_{i2}$ and $\vec{K}'_{i1} \fatsemi \vec{K}'_{i2}$, respectively, for all i, $1 \leq i \leq k$. According to Remark 7.14 we have that

$$seq(\vec{K}_1, \ldots, \vec{K}_k) = seq(\vec{K}_{11}, \ldots, \vec{K}_{k1}) \fatsemi seq(\vec{K}_{12}, \ldots, \vec{K}_{k2})$$

and

$$seq(\vec{K}'_1, \ldots, \vec{K}'_k) = seq(\vec{K}'_{11}, \ldots, \vec{K}'_{k1}) \fatsemi seq(\vec{K}'_{12}, \ldots, \vec{K}'_{k2}).$$

Using the definition of the $\longrightarrow\!\!\!\!\!\!\rightarrow$ relation given in Figure 7.5, and the induction hypotheses, it follows that $\vec{K} \longrightarrow\!\!\!\!\!\!\rightarrow \vec{K}'$. Next, assume that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. Then, we can write $\vec{K}_i$ and $\vec{K}'_i$ as $\langle l_s, \Psi_{si} \rangle$ `if` C `then` $\vec{K}_{ci}$ `else` $\vec{K}_{ai}$ `endif` $\langle l_f, \Psi_{fi} \rangle$ and $\langle l_s, \Psi_{si} \rangle$ `if` C `then` $\langle first(P_a), filter(C, \Psi_{si}) \rangle \fatsemi \vec{K}'_{ci}$ `else` $\langle first(P_c), filter(\neg C, \Psi_{si}) \rangle \fatsemi \vec{K}'_{ai}$ `endif` $\langle l_f, \vec{K}_{ci}|_{last(P_c)} \cup \vec{K}_{ai}|_{last(P_a)} \rangle$, respectively, with $\vec{K}_{ci} \longrightarrow\!\!\!\!\!\!\rightarrow \vec{K}'_{ci}$ and $\vec{K}_{ai} \longrightarrow\!\!\!\!\!\!\rightarrow \vec{K}'_{ai}$, for all i, $1 \leq i \leq k$. According to Remarks 7.14 and 7.15, we have that

$$\begin{array}{rcl} seq(\vec{K}'_1, \ldots, \vec{K}'_k) & = & seq(\Psi_{s1}, \ldots, \Psi_{sk}) \\ & & \quad \texttt{if } C \texttt{ then } seq(\vec{K}_{c1}, \ldots, \vec{K}_{ck}) \\ & & \quad \texttt{else } seq(\vec{K}_{a1}, \ldots, \vec{K}_{ak}) \\ & & \quad \texttt{endif} \\ & & seq(\Psi_{f1}, \ldots, \Psi_{fk}) \end{array}$$

and

$$\begin{array}{rcl} seq(\vec{K}_1, \ldots, \vec{K}_k) & = & \langle l_s, seq(\Psi_{s1}, \ldots, \Psi_{sk}) \rangle \\ & & \quad \texttt{if } C \texttt{ then } \langle first(P_a), filter(C, seq(\Psi_{s1}, \ldots, \Psi_{s_k})) \rangle \\ & & \qquad \fatsemi seq(\vec{K}'_{c1}, \ldots, \vec{K}'_{ck}) \\ & & \quad \texttt{else } \langle first(P_c), filter(\neg C, seq(\Psi_{s1}, \ldots, \Psi_{s_k})) \rangle \\ & & \qquad \fatsemi seq(\vec{K}'_{a1}, \ldots, \vec{K}'_{ak}) \\ & & \quad \texttt{endif} \\ & & \langle l_f, seq(\vec{K}_{c1}, \ldots, \vec{K}_{ck})|_{last(P_c)} \cup seq(\vec{K}_{a1}, \ldots, \vec{K}_{ak})|_{last(P_a)} \rangle. \end{array}$$

Using the definition of the $\longrightarrow\!\!\!\!\!\!\rightarrow$ relation given in Figure 7.5 and the induction hypotheses, it follows that $\vec{K} \longrightarrow\!\!\!\!\!\!\rightarrow \vec{K}'$. Finally, assume that P is the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$. Then, we can write $\vec{K}_i$ and $\vec{K}'_i$ as $\langle l_s, \Psi_{si} \rangle$ `while` C

$\texttt{do}\ \vec{K}_{bi}\ \texttt{endwhile}\ \langle l_f, \Psi_{fi}\rangle$ and $\langle l_s, \Psi_{si}\rangle\ \texttt{while}\ C\ \texttt{do}\ \mathit{filter}(C, \mathit{before}(\Psi_{si}, \vec{K}_{bi}|_{\mathit{last}(P_b)}))\fatsemi \vec{K}'_{bi}\ \ \texttt{endwhile}\ \ \langle l_f, \mathit{filter}(\neg C, \Psi_{si} \cup \mathit{collect}(\vec{K}_{bi}|_{\mathit{last}(P_b)}))\rangle$, respectively, with $\vec{K}_{bi} \longrightarrow \vec{K}'_{bi}$, for all i, $1 \leq i \leq k$. According to Remarks 7.14 and 7.15, we have that

$$\begin{aligned} \mathit{seq}(\vec{K}_1, \ldots, \vec{K}_k) \ = \ & \langle l_s, \mathit{seq}(\Psi_{s1}, \ldots, \Psi_{sk})\rangle \\ & \texttt{while}\ C\ \texttt{do} \\ & \quad \mathit{seq}(\vec{K}_{b1}, \ldots, \vec{K}_{bk}) \\ & \texttt{endwhile} \\ & \langle l_f, \mathit{seq}(\Psi_{f1}, \ldots, \Psi_{fk})\rangle \end{aligned}$$

and

$$\begin{aligned} \mathit{seq}(\vec{K}'_1, \ldots, \vec{K}'_k) \ = \ & \langle l_s, \mathit{seq}(\Psi_{s1}, \ldots, \Psi_{sk})\rangle \\ & \texttt{while}\ C\ \texttt{do} \\ & \quad \mathit{filter}(C, \mathit{before}(\mathit{seq}(\Psi_{s1}, \ldots, \Psi_{sk}), \mathit{seq}(\vec{K}_{bi}, \ldots, \vec{K}_{bi})|_{\mathit{last}(P_b)})) \\ & \qquad \fatsemi \mathit{seq}(\vec{K}'_{bi}, \ldots, \vec{K}'_{bi}) \\ & \texttt{endwhile} \\ & \langle l_f, \mathit{filter}(\neg C, \mathit{seq}(\Psi_{s1}, \ldots, \Psi_{sk}) \cup \mathit{collect}(\mathit{seq}(\vec{K}_{b1}, \ldots, \vec{K}_{bk})|_{\mathit{last}(P_b)}))\rangle \end{aligned} .$$

Using the definition of the $\longrightarrow$ relation given in Figure 7.5 and the induction hypotheses, it follows that $\vec{K} \longrightarrow \vec{K}'$. Using the definition of the $\longrightarrow$ relation given in Figure 7.5 and the induction hypotheses, it follows that $\vec{K} \longrightarrow \vec{K}'$. $\square$

7.17 Proposition Let P be a program. The following two statements hold.

a) Denote by $\vec{K}_{P,\perp}$ the empty configuration corresponding to P. Then, $\vec{K}_{P,\perp} \longrightarrow \vec{K}_{P,\perp}$.

b) Let θ be a terminal trace w.r.t P, and denote by $\vec{K}$ the configuration such that $\mathit{rep}_P(\theta) = \vec{K}$. Then $K \longrightarrow K$.

Proof: The proof is by structural induction on program P. For the base case, assume first that P is a skip program, and denote by l_s and l_f the first and last labels of P, respectively. Then, $\vec{K}_{P,\perp} = \langle l_s, \lambda\langle\rangle\,.\,\emptyset\rangle\ \texttt{skip}\ \langle l_f, \lambda\langle\rangle\,.\,\emptyset\rangle$. Also, given a terminal trace w.r.t. P, $\theta = \langle l_s, \sigma\rangle\langle l_f, \sigma\rangle$ for some environment $\sigma \in \mathbf{Env}$, according to Definition 5.1 the representation of θ is the configuration $\vec{K} = \langle l_s, \lambda\langle\rangle\,.\,\{\sigma\}\rangle\ \texttt{skip}\ \langle l_f, \lambda\langle\rangle\,.\,\{\sigma\}\rangle$. It is easy to verify that the relations $\vec{K}_{P,\perp} \longrightarrow \vec{K}_{P,\perp}$ and $\vec{K} \longrightarrow \vec{K}$, as defined in Figure 7.5, hold. Assume now that P is the assignment $\langle l_s\rangle\, x := E\, \langle l_f\rangle$, where l_s and l_f are labels, $x \in \mathbf{Var}$, and $E \in \mathbf{Expr}$. In this case $\vec{K}_{P,\perp} = \langle l_s, \lambda\langle\rangle\,.\,\emptyset\rangle\ x := E\ \langle l_f, \lambda\langle\rangle\,.\,\emptyset\rangle$ and $\vec{K} = \langle l_s, \lambda\langle\rangle\,.\,\{\sigma\}\rangle\ \texttt{skip}\ \langle l_f, \lambda\langle\rangle\,.\,\{\sigma[x \mapsto E(\sigma)]\}\rangle$. Again, it is easy to verify that the two relations hold.

We continue with the proof of the inductive cases. Assume first that $P = P_1 \fatsemi P_2$ is a sequence program. Then, $\vec{K}_{P,\perp} = \vec{K}_{P_1,\perp} \fatsemi \vec{K}_{P_2,\perp}$, where $\vec{K}_{P_1,\perp}$ and $\vec{K}_{P_2,\perp}$ are the empty configurations corresponding to programs P_1 and P_2, respectively. Also, if θ is a terminal trace w.r.t P, then according to Proposition 7.1 $\theta \sim \underbrace{t_1}_{P_1} \xrightarrow{P_1} \underbrace{\langle \mathit{last}(P_1), \sigma\rangle t_2}_{P_2}$. The trace $t_1\langle \mathit{last}(P_1), \sigma\rangle$ is terminal w.r.t. P_1, while the trace $\langle \mathit{last}(P_1), \sigma\rangle t_2$ is terminal w.r.t. P_2. Denote by $\vec{K}_1$ and $\vec{K}_2$ the configurations that represent $t_1\langle \mathit{last}(P_1), \sigma\rangle$ and $\langle \mathit{last}(P_1), \sigma\rangle t_2$, respectively. Proposition 7.9 entails that $\vec{K} = \vec{K}_1 \fatsemi \vec{K}_2$. According to the induction hypothesis, we have that $\vec{K}_{P_1,\perp} \longrightarrow \vec{K}_{P_1,\perp}$, $\vec{K}_{P_2,\perp} \longrightarrow \vec{K}_{P_2,\perp}$, $\vec{K}_1 \longrightarrow \vec{K}_1$, and $\vec{K}_2 \longrightarrow \vec{K}_2$. It follows from the definition of the $\longrightarrow$ relation given in Figure 7.5 that the two relations hold.

Assume now that P is the `if` program $\langle l_s\rangle\ \texttt{if}\ C\ \texttt{then}\ P_c\ \texttt{else}\ P_a\ \texttt{endif}\ \langle l_f\rangle$. Then, $\vec{K}_{P,\perp} = \langle l_s, \lambda\langle\rangle\,.\,\emptyset\rangle\ \texttt{if}\ C\ \texttt{then}\ \vec{K}_{P_c,\perp}\ \texttt{else}\ \vec{K}_{P_a,\perp}\ \texttt{endif}\ \langle l_f, \lambda\langle\rangle\,.\,\emptyset\rangle$. According to the induction hypothesis, $\vec{K}_{P_c,\perp} \longrightarrow \vec{K}_{P_c,\perp}$ and $\vec{K}_{P_a,\perp} \longrightarrow \vec{K}_{P_a,\perp}$.

It follows from the definition of the $\longrightarrow$ relation that $\vec{K}_{P,\perp} \longrightarrow \vec{K}_{P,\perp}$. If θ is a terminal trace w.r.t P, then, according to Proposition 7.2 we have two sub-cases: either $\theta \sim \langle l_s, \sigma_s\rangle \xrightarrow{P\swarrow} \underbrace{t}_{P_c} \xrightarrow{P\nearrow} \langle l_f, \sigma_f\rangle$, where $\sigma_s \models C$ and t is a terminal trace w.r.t. P_c, or $\theta \sim \langle l_s, \sigma_s\rangle \xrightarrow{P\searrow} \underbrace{t}_{P_a} \xrightarrow{P\nwarrow} \langle l_f, \sigma_f\rangle$, where $\sigma_s \models \neg C$ and t is a terminal trace w.r.t. P_a. We shall prove only the first sub-case, since the proof of the second one is similar. Denote by $\vec{K}_c$ the configuration representing the trace t w.r.t. P_c. Then, according to Proposition 7.10 we have $\vec{K} = \langle l_s, \lambda\langle\rangle\,.\,\{\sigma_s\}\rangle\ \texttt{if}\ C\ \texttt{then}\ \vec{K}_c\ \texttt{else}\ \vec{K}_{P_a,\perp}\ \texttt{endif}\ \langle l_f, \lambda\langle\rangle\,.\,\{\sigma_f\}\rangle$. The induction hypothesis states that $\vec{K}_c \longrightarrow \vec{K}_c$ and $\vec{K}_{P_a,\perp} \longrightarrow \vec{K}_{P_a,\perp}$. Moreover, since $\sigma_s \models C$, we have that $\mathit{filter}(C, \lambda\langle\rangle\,.\,\{\sigma_s\}) = \lambda\langle\rangle\,.\,\{\sigma_s\}$ and $\mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\{\sigma_s\}) = \lambda\langle\rangle\,.\,\emptyset$. This entails that $\vec{K}_c = \langle \mathit{first}(K), \mathit{filter}(C, \lambda\langle\rangle\,.\,\{\sigma_s\})\rangle \fatsemi \vec{K}_c$ and $\vec{K}_{P_a,\perp} = \langle \mathit{first}(K), \mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\{\sigma_s\})\rangle \fatsemi \vec{K}_{P_a,\perp}$ from which it follows that $\vec{K} \longrightarrow \vec{K}$.

Finally, assume that P is the `while` program $\langle l_s\rangle\ \texttt{while}\ C\ \texttt{do}\ P_b\ \texttt{endwhile}\ \langle l_f\rangle$. Then, $\vec{K}_{P,\perp} = \langle l_s, \lambda\langle\rangle\,.\,\emptyset\rangle\ \texttt{while}\ C\ \texttt{do}\ \vec{K}_{P_b,\perp}\ \texttt{endwhile}\ \langle l_f, \lambda\langle\rangle\,.\,\emptyset\rangle$. According to the induction hypothesis, $\vec{K}_{P_b,\perp} \longrightarrow \vec{K}_{P_b,\perp}$. It follows from the definition of the $\longrightarrow$ relation that $\vec{K}_{P,\perp} \longrightarrow \vec{K}_{P,\perp}$. If θ is a terminal trace w.r.t P, then, according to Proposition 7.3 we have two sub-cases: either $\theta \sim \langle l_s, \sigma\rangle \xrightarrow{P\curvearrowright} \langle l_f, \sigma\rangle$, where $\sigma \models \neg C$, or $\theta \sim \langle l_s, \sigma_s\rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle l_f, \sigma_f\rangle$, where trace segments $t_1, \ldots, t_k$ are terminal traces w.r.t. P_b. For the first sub-case, according to Proposition 7.11 we have $\vec{K} = \langle l_s, \lambda\langle\rangle\,.\,\{\sigma\}\rangle\ \texttt{while}\ C\ \texttt{do}\ \vec{K}_{P_b,\perp}\ \texttt{endwhile}\ \langle l_f, \lambda\langle\rangle\,.\,\{\sigma\}\rangle$. The induction hypothesis states that $\vec{K}_{P_b,\perp} \longrightarrow \vec{K}_{P_b,\perp}$. Moreover, since $\sigma \models \neg C$, we have that $\mathit{filter}(C, \mathit{before}(\lambda\langle\rangle\,.\,\{\sigma\}, \lambda\langle\rangle\,.\,\emptyset)) = \lambda\langle\rangle\,.\,\emptyset$ and by noting that $\vec{K}_{P_b,\perp}|_{\mathit{last}(P_b)} = \lambda\langle\rangle\,.\,\emptyset$, it follows that $\mathit{filter}(C, \mathit{before}(\lambda\langle\rangle\,.\,\{\sigma\}, \vec{K}_{P_b,\perp}|_{\mathit{last}(P_b)})) \fatsemi \vec{K}_{P_b,\perp} = \vec{K}_{P_b,\perp}$. We also note that $\mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\{\sigma\} \cup \mathit{collect}(\vec{K}_{P_b,\perp}|_{\mathit{last}(P_b)})) = \lambda\langle\rangle\,.\,\{\sigma\}$. By applying the

definition of the transition relation given in Figure 7.5 it follows that $\vec{K} \longrightarrow \vec{K}$.

For the second sub-case, assume that we have $\theta \sim \langle l_s, \sigma_s\rangle \xrightarrow{P\searrow} \underbrace{t_1}_{P_b} \xrightarrow{P\circlearrowleft} \cdots \xrightarrow{P\circlearrowleft} \underbrace{t_k}_{P_b} \xrightarrow{P\nwarrow} \langle l_f, \sigma_f\rangle$, where trace segments $t_1, \ldots, t_k$ are terminal traces w.r.t. P_b. Given a rank i, $1 \leq i < k$, we have that the environment of the last state of segment t_i is the same as the environment of the first state of segment t_{i+1}. Also, the environment of the first state of t_1 is σ_s, while the environment of the last state of t_k is σ_f. Denote by $\vec{K}_i$ the singleton configurations that represent segments t_i w.r.t. P_b, $1 \leq i \leq k$, and by $\vec{K}_{seq}$ the sequencing $seq(\vec{K}_1, \ldots, \vec{K}_k)$. The above property entails that $\vec{K}_1|_{\mathit{first}(P_b)} = \lambda\langle\rangle\,.\,\{\sigma_s\}$, $\vec{K}_k|_{\mathit{last}(P_b)} = \lambda\langle\rangle\,.\,\{\sigma_f\}$, and $\vec{K}_i|_{\mathit{last}(P_b)} = \vec{K}_{i+1}|_{\mathit{first}(P_b)}$, for all i, $1 \leq i < k$. From this it follows that $\mathit{filter}(C, \mathit{before}(\lambda\langle\rangle\,.\,\{\sigma_s\}, \vec{K}_{seq}|_{\mathit{last}(P_b)})) \fatsemi \vec{K}_{seq} = \vec{K}_{seq}$. Since all segments t_i are terminal, for all i, $1 \leq i \leq k$, from the induction hypothesis we have that $\vec{K}_i \longrightarrow \vec{K}_i$, which in turn entails, according to Proposition 7.16 that $\vec{K}_{seq} \longrightarrow \vec{K}_{seq}$. We also have that $\mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\{\sigma_s\} \cup \mathit{collect}(\vec{K}_{seq}|_{\mathit{last}(P_b)})) = \lambda\langle\rangle\,.\,\{\sigma_f\}$. By applying the definition of the transition relation given in Figure 7.5, it follows that $\vec{K} \longrightarrow \vec{K}$. □

7.18 Proposition Consider a labeled program P and let $\theta_1, \theta_2 \in \overrightarrow{P}$ be two traces such that θ_1 is the longest proper prefix of θ_2. Let $\vec{K}_1$ and $\vec{K}_2$ be two singleton configurations such that $rep_P(\theta_1) = \vec{K}_1$ and $rep_P(\theta_2) = \vec{K}_2$. Then, $\vec{K}_1 \longrightarrow \vec{K}_2$.

Proof: The proof is by structural induction on the program P. We have two base cases, when P is either a skip statement or an assignment. Assume first that P is the skip statement $\langle l_s\rangle\,\texttt{skip}\,\langle l_f\rangle$. Then, Definition 5.1 entails that $\theta_1 = \langle l_s, \sigma\rangle$, and $\theta_2 = \langle l_s, \sigma\rangle\langle l_f, \sigma\rangle$ for some environment $\sigma \in \mathbf{Env}$. According to Relation 7.12 we have that $rep_P(\theta_1) = \langle l_s, \lambda\langle\rangle\,.\,\{\sigma\}\rangle\,\texttt{skip}\,\langle l_f, \lambda\langle\rangle\,.\,\emptyset\rangle = \vec{K}_1$,

and $rep_P(\theta_2) = \langle l_s, \lambda\langle\rangle \,.\, \{\sigma\}\rangle \,\texttt{skip}\, \langle l_f, \lambda\langle\rangle \,.\, \{\sigma\}\rangle = \vec{K}_2$. It is easy to verify that the relation $\vec{K}_1 \longrightarrow \vec{K}_2$, as defined in Figure 7.5 holds. Assume now that P is the assignment $\langle l_s\rangle\, x := E\, \langle l_f\rangle$, where l_s and l_f are labels, $x \in \mathbf{Var}$, and $E \in \mathbf{Expr}$. Definition 5.1 entails that $\theta_1 = \langle l_s, \sigma\rangle$, and $\theta_2 = \langle l_s, \sigma\rangle\langle l_f, \sigma[x \mapsto E(\sigma)]\rangle$ for some environment $\sigma \in \mathbf{Env}$. According to Relation 7.12, we have that $rep_P(\theta_1) = \langle l_s, \lambda\langle\rangle \,.\, \{\sigma\}\rangle\, x := E\, \langle l_f, \lambda\langle\rangle \,.\, \emptyset\rangle = \vec{K}_1$, and $rep_P(\theta_2) = \langle l_s, \lambda\langle\rangle \,.\, \{\sigma\}\rangle\, x := E\, \langle l_f, \lambda\langle\rangle \,.\, \{\sigma[x \mapsto E(\sigma)]\}\rangle = \vec{K}_2$. It is easy to verify that the relation $\vec{K}_1 \longrightarrow \vec{K}_2$, as defined in Figure 7.5, holds.

We now proceed with the induction cases. Assume first that $P = P_1 \,;\, P_2$ is a sequence program. According to Proposition 7.1, we have the following cases for the traces θ_1 and θ_2. $\square$

8.2 Proposition Let P be a labeled program, and S a set of states of P. Denote by S_0 the set $\{\langle l, \sigma\rangle \,|\, \langle l, \sigma\rangle \in S \text{ and } l = \mathit{first}(P)\}$. Assume that the representation of S is a configuration $\overline{K}$. Then $(T_P \cup S_0)(S)$ is represented by $T(\overline{K})$.

Proof: We need to prove that for every label $l \in \mathit{labels}(P)$ we have $T(\overline{K})|_l = \{\langle l', \sigma'\rangle \,|\, \langle l', \sigma'\rangle \in (T_P \cup S_0)(S) \text{ and } l' = l\}$. The proof is by induction on the structure of P. Assume first that P is the skip program $\langle l_s\rangle\, \texttt{skip}\, \langle l_f\rangle$. Then, $(T_P \cup S_0)(S) = S_0 \cup \{\langle l_f, \sigma\rangle \,|\, \langle l_s, \sigma\rangle \in S_0\}$. On the other hand, $T(\overline{K}) = \langle l_s, \{\sigma \,|\, \langle l_s, \sigma\rangle \in S_0\}\rangle \,\texttt{skip}\, \langle l_f, \{\sigma \,|\, \langle l_s, \sigma\rangle \in S_0\}\rangle$. Clearly, $(T_P \cup S_0)(S)$ is represented by $T(\overline{K})$. Next, assume that P is the assignment $\langle l_s\rangle\, x := E\, \langle l_f\rangle$. Then, $(T_P \cup S_0)(S) = S_0 \cup \{\langle l_f, \sigma\rangle \,|\, \text{there exists } \sigma' \text{ s.t. } \langle l_s, \sigma'\rangle \in S_0 \text{ and } \sigma = \sigma'[x \mapsto E(\sigma')]\}$. On the other hand, $T(\overline{K}) = \langle l_s, \{\sigma \,|\, \langle l_s, \sigma\rangle \in S_0\}\rangle\ x := E\ \langle l_f, \{\sigma \,|\, \text{there exists } \sigma' \text{ s.t. } \langle l_s, \sigma'\rangle \in S_0 \text{ and } \sigma = \sigma'[x \mapsto E(\sigma')]\}\rangle$. It can be easily observed that $(T_P \cup S_0)(S)$ is represented by $T(\overline{K})$.

Assume now that P is the sequence program $P_1 \fatsemi P_2$. In this case, configuration $\overline{K}$ can be written as $\overline{K}_1 \fatsemi \overline{K}_2$, where $|\overline{K}_1| = P_1$ and $|\overline{K}_2| = P_2$. Given a label $l \in \mathit{labels}(P)$, we have that either $l \in \mathit{labels}(P_1)$, or $l \in \mathit{labels}(P_2) \backslash \{\mathit{last}(P_1)\}$. Denote by S_1 and S_2 the subsets of S that contain all the states with labels from P_1 and P_2, respectively. Then, S_1 is represented by $\overline{K}_1$ and S_2 is represented by $\overline{K}_2$. Denote by S_{20} the set of states whose label is $\mathit{first}(P_2)$. According to the induction hypothesis, $(T_{P_1} \cup S_0)(S_1)$ is represented by $T(\overline{K}_1)$, and $(T_{P_2} \cup S_{20})(S_2)$ is represented by $T(\overline{K}_2)$. Moreover, according to the definition of T_P (Definition 5.9) we have that $(T_P \cup S_0)(S) = (T_{P_1} \cup S_0)(S_1) \cup ((T_{P_2} \cup S_{20})(S_2) \backslash S_{20})$. Now, if the label l is a member of $\mathit{labels}(P_1)$, we have that $T(\overline{K})|_l = T(\overline{K}_1)|_l = \{\langle l', \sigma' \rangle \,|\, \langle l', \sigma' \rangle \in (T_{P_1} \cup S_0)(S_1)$ and $l' = l\} = \{\langle l', \sigma' \rangle \,|\, \langle l', \sigma' \rangle \in (T_P \cup S_0)(S)$ and $l' = l\}$. If, on the other hand, l is a member of $\mathit{labels}(P_2) \backslash \{\mathit{last}(P_1)\}$, we have that $T(\overline{K})|_l = T(\overline{K}_2)|_l = \{\langle l', \sigma' \rangle \,|\, \langle l', \sigma' \rangle \in ((T_{P_2} \cup S_{20})(S_2) \backslash S_{20})$ and $l' = l\ \} = \{\langle l', \sigma' \rangle \,|\, \langle l', \sigma' \rangle \in (T_P \cup S_0)(S)$ and $l' = l\ \}$.

Assume now that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. For convenience, we introduce the following notations: $S_c \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l \in \mathit{labels}(P_c)\}$, $S_a \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l \in \mathit{labels}(P_a)\}$, $S_{c0} \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l = \mathit{first}(P_c)\}$, $S_{c1} \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l = \mathit{last}(P_c)\}$, $S_{a0} \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l = \mathit{first}(P_a)\}$, and $S_{a1} \stackrel{not}{=} \{\langle l, \sigma \rangle \,|\, \langle l, \sigma \rangle \in S$ and $l = \mathit{last}(P_a)\}$. Then we have $(T_P \cup S_0)(S) = S_0 \cup \{\langle \mathit{first}(P_c), \sigma \rangle \,|\, \langle l_s, \sigma \rangle \in S_0$ and $\sigma \models C\} \cup \{\langle \mathit{first}(P_a), \sigma \rangle \,|\, \langle l_s, \sigma \rangle \in S_0$ and $\sigma \models \neg C\} \cup ((T_{P_c} \cup S_{0c})(S_c) \backslash S_{0c}) \cup ((T_{P_a} \cup S_{0a})(S_a) \backslash S_{0a}) \cup \{\langle l_f, \sigma \rangle \,|\, \langle \mathit{last}(P_c), \sigma \rangle \in S_{1c}\} \cup \{\langle l_f, \sigma \rangle \,|\, \langle \mathit{last}(P_a), \sigma \rangle \in S_{1a}\}$. Consider now a label $l \in \mathit{labels}(P)$. We have six sub-cases: either $l = l_s$, or $l = \mathit{first}(P_c)$, or $l = \mathit{first}(P_a)$, or $l \in \mathit{labels}(P_c) \backslash \{\mathit{first}(P_c)\}$, or $l \in \mathit{labels}(P_a) \backslash \{\mathit{first}(P_a)\}$, or $l = l_f$. If $l = l_s$, then $T(\overline{K})|_l = S_0 = \{\langle l', \sigma' \rangle \,|\, \langle l', \sigma' \rangle \in (T_P \cup S_0)(S)$ and $l' = l_s\}$. If $l = \mathit{first}(P_c)$, then $T(\overline{K})|_l = \{\langle \mathit{first}(P_c), \sigma \rangle \,|\, \langle l_s, \sigma \rangle \in S_0$ and $\sigma \models C\} =$

$\{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = \mathit{first}(P_c)\}$. If $l = \mathit{first}(P_a)$, then $T(\overline{K})|_l = \{\langle \mathit{first}(P_a), \sigma\rangle \mid \langle l_s, \sigma\rangle \in S_0$ and $\sigma \models \neg C\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = \mathit{first}(P_a)\}$. If $l \in \mathit{labels}(P_c) \backslash \{\mathit{first}(P_c)\}$, then $T(\overline{K})|_l = T(\overline{K}_c)|_l$. According to the induction hypothesis, $T(\overline{K}_c)|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_c} \cup S_{c0})(S_c)$ and $l' = l\}$. It follows that $T(\overline{K})|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_c} \cup S_{c0})(S_1)$ and $l' = l\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = l\}$. If $l \in \mathit{labels}(P_a) \backslash \{\mathit{first}(P_a)\}$, then $T(\overline{K})|_l = T(\overline{K}_a)|_l$. According to the induction hypothesis, $T(\overline{K}_a)|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_a} \cup S_{a0})(S_a)$ and $l' = l\}$. It follows that $T(\overline{K})|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_a} \cup S_{a0})(S_a)$ and $l' = l\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = l\}$. If $l = l_f$, then $T(\overline{K})|_l = \{\langle l_f, \sigma\rangle \mid \langle \mathit{last}(P_c), \sigma\rangle \in S_{1c}\} \cup \{\langle l_f, \sigma\rangle \mid \langle \mathit{last}(P_a), \sigma\rangle \in S_{1a}\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = \mathit{last}(P_a)\}$.

Finally, assume that P is the `while` program $\langle l_s\rangle$ `while` C `then` P_b `endwhile` $\langle l_f\rangle$. For convenience, we introduce the following notations: $S_b \stackrel{not}{=} \{\langle l, \sigma\rangle \mid \langle l, \sigma\rangle \in S$ and $l \in \mathit{labels}(P_b)\}$, $S_{b0} \stackrel{not}{=} \{\langle l, \sigma\rangle \mid \langle l, \sigma\rangle \in S$ and $l = \mathit{first}(P_b)\}$, and $S_{b1} \stackrel{not}{=} \{\langle l, \sigma\rangle \mid \langle l, \sigma\rangle \in S$ and $l = \mathit{last}(P_b)\}$. Then we have $(T_P \cup S_0)(S) = S_0 \cup (\{\langle \mathit{first}(P_b), \sigma\rangle \mid \langle l_s, \sigma\rangle \in S_0$ and $\sigma \models C\} \cup \{\langle \mathit{first}(P_b), \sigma\rangle \mid \langle \mathit{last}(P_b), \sigma\rangle \in S_{b1}$ and $\sigma \models C\}) \cup (T_{P_b} \cup S_{b0})(S_b) \cup (\{\langle l_f, \sigma\rangle \mid \langle l_s, \sigma\rangle \in S_0$ and $\sigma \models \neg C\} \cup \{\langle l_f, \sigma\rangle \mid \langle \mathit{last}(P_b), \sigma\rangle \in S_{b1}$ and $\sigma \models \neg C\})$. Consider now a label $l \in \mathit{labels}(P)$. We have four sub-cases: either $l = l_s$, or $l = \mathit{first}(P_b)$, or $l \in \mathit{labels}(P_b) \backslash \{\mathit{first}(P_b)\}$, or $l = l_f$. If $l = l_s$, then $T(\overline{K})|_l = S_0 = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = l_s\}$. If $l = \mathit{first}(P_b)$, then $T(\overline{K})|_l = \{\langle \mathit{first}(P_b), \sigma\rangle \mid \langle l_s, \sigma\rangle \in S_0$ and $\sigma \models C\} \cup \{\langle \mathit{first}(P_b), \sigma\rangle \mid \langle \mathit{last}(P_b), \sigma\rangle \in S_{b1}$ and $\sigma \models C\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = \mathit{first}(P_b)\}$. If $l \in \mathit{labels}(P_b) \backslash \{\mathit{first}(P_b)\}$, then $T(\overline{K})|_l = T(\overline{K}_b)|_l$. According to the induction hypothesis, $T(\overline{K}_b)|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_b} \cup S_{b0})(S_b)$ and $l' = l\}$. It follows that $T(\overline{K})|_l = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_{P_b} \cup S_{b0})(S_b)$ and $l' = l\} = \{\langle l', \sigma'\rangle \mid \langle l', \sigma'\rangle \in (T_P \cup S_0)(S)$ and $l' = l\}$. The last sub-case is $l = l_f$. In this case

we have $T(\overline{K})|_l = \{\langle l_f, \sigma\rangle \,|\, \langle l_s, \sigma\rangle \in S_0 \text{ and } \sigma \models \neg C\} \cup \{\langle l_f, \sigma\rangle \,|\, \langle \mathit{last}(P_b), \sigma\rangle \in S_{b1} \text{ and } \sigma \models \neg C\} = \{\langle l', \sigma'\rangle \,|\, \langle l', \sigma'\rangle \in (T_P \cup S_0)(S) \text{ and } l' = \mathit{last}(P)\}$. □

8.3 Proposition Let P be a program, Σ_0 a set of start environments, and denote by $\overline{K}_0$ the configuration such that $|\overline{K}_0| = P$, $\overline{K}_0|_{\mathit{first}(P)} = \Sigma_0$, and $\overline{K}_0|_l = \emptyset$ for all labels $l \in \mathit{labels}(P)\backslash\{\mathit{first}(P)\}$. Then, $T^\omega(\overline{K}_0)$ represents the collecting semantics of P w.r.t. Σ_0.

Proof: Let us denote by $\overline{K}^{CS}$ the configuration that represents the set of states $CS_P = \mathit{lub}\{T_P^n(S_0) \,|\, n \in \mathbb{N}\}$, and by S_0 the set $\{\langle \mathit{first}(P), \sigma\rangle \,|\, \sigma \in \Sigma_0\}$. We need to show that $\overline{K}^{CS}$ is $T^\omega(\overline{K}_0) = \mathit{lub}(\{T^n(\overline{K}_0) \,|\, n \in \mathbb{N}\})$. We first show by induction that for all n we have that the configuration $\overline{K}_n = T^n(\overline{K_0})$ represents the set of states $T_P^n(S_0)$. The base case is trivially true. For the induction case, assume that the property of $\overline{K}_n$ representing the set of states $T_P^n(S_0)$ holds. Then, according to Proposition 8.2 we have that $\overline{K}_{n+1} = T(\overline{K}_n)$ represents $T_P(T_P^n(S_0)) = T_P^{n+1}(S_0)$. We now show that $\overline{K}^{CS}$ is $\mathit{lub}(\{T^n(\overline{K}_0) \,|\, n \in \mathbb{N}\})$, by first showing that $\overline{K}^{CS}$ is an upper bound, and then that it is the least one. Indeed, from $T_P^n(S_0) \subseteq CS_P$, and $T_P^n(S_0)$ and CS_P being represented by $\overline{K}_n$ and $\overline{K}^{CS}$, respectively, we have that $\overline{K}_n \preccurlyeq \overline{K}^{CS}$, for all $n \in \mathbb{N}$. On the other hand, let $\overline{K}$ be an upper bound of the set $\{T^n(\overline{K}_0) \,|\, n \in \mathbb{N}\}$, and denote by S the set of states represented by $\overline{K}$ (according to Proposition 8.1 S is unique). From $\overline{K}_n \preccurlyeq \overline{K}$ it follows that $T_P^n(S_0) \subseteq S$, that is, S is an upper bound of $\{T^n(\overline{K}_0) \,|\, n \in \mathbb{N}\}$. As a result, we have that $T^\omega(\overline{K}_0) \subseteq S$, and it follows that $\overline{K}^{CS} \preccurlyeq \overline{K}$. □

8.4 Proposition Given a set of collective configurations with the same skeleton $\overline{\Gamma}$, we have that $T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}) = \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K})$.

Proof: Let us denote by P the program that is the skeleton of configurations in $\overline{\Gamma}$. The proof is by induction on the structure of P. Assume first that P is the skip program $\langle l_s \rangle \texttt{skip} \langle l_f \rangle$. Then we have that $\overline{K} = \langle l_s, \overline{K}|_{l_s} \rangle \texttt{ skip } \langle l_f, \overline{K}|_{l_f} \rangle$, for all $\overline{K} \in \overline{\Gamma}$. It follows that

$$\begin{aligned} T(\bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}) &= T(\bigcup_{\overline{K} \in \overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \texttt{ skip } \langle l_f, \overline{K}|_{l_f} \rangle)) \\ &= T(\langle l_s, \bigcup_{\overline{K} \in \overline{\Gamma}} |_{l_s} \rangle \texttt{skip} \langle l_f, \bigcup_{\overline{K} \in \overline{\Gamma}} |_{l_f} \rangle \\ &= \bigcup_{\overline{K} \in \overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \texttt{ skip } \langle l_f, \overline{K}|_{l_s} \rangle) \\ &= \bigcup_{\overline{K} \in \overline{\Gamma}} T(\langle l_s, \overline{K}|_{l_s} \rangle \texttt{ skip } \langle l_f, \overline{K}|_{l_f} \rangle) \\ &= T(\overline{K}), \end{aligned}$$

for all $\overline{K} \in \overline{\Gamma}$. Assume now that P is the assignment $\langle l_s \rangle \, x := E \, \langle l_f \rangle$. Then we have that $\overline{K} = \langle l_s, \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \overline{K}|_{l_f} \rangle$, for all $\overline{K} \in \overline{\Gamma}$. It follows that

$$\begin{aligned} T(\bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}) &= T(\bigcup_{\overline{K} \in \overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \overline{K}|_{l_f} \rangle)) \\ &= T(\langle l_s, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_f} \rangle) \\ &= T(\langle l_s, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_f} \rangle) \\ &= \langle l_s, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}|_{l_s}[x \mapsto E] \rangle \\ &= \bigcup_{\overline{K} \in \overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \overline{K}|_{l_s}[x \mapsto E] \rangle) \\ &= \bigcup_{\overline{K} \in \overline{\Gamma}} T(\langle l_s, \overline{K}|_{l_s} \rangle \, x := E \, \langle l_f, \overline{K}|_{l_f} \rangle) \\ &= \bigcup_{\overline{K} \in \overline{\Gamma}} T(\overline{K}), \end{aligned}$$

for all $\overline{K} \in \overline{\Gamma}$. Next, assume that P is the sequence program $P_1 \fatsemi P_2$. Then, given a $\overline{K} \in \overline{\Gamma}$, we denote by $\overline{K}^{fst}$ and $\overline{K}^{snd}$ the sub-configurations of $\overline{K}$ whose skeletons are P_1 and P_2, respectively. Obviously, $\overline{K} = \overline{K}^{fst} \fatsemi \overline{K}^{snd}$, for all $\overline{K} \in \overline{\Gamma}$. Moreover, from the induction hypothesis we have that $T(\bigcup_{\overline{K} \in \overline{\Gamma}} \overline{K}^{fst}) = \bigcup_{\overline{K} \in \overline{\Gamma}} T(\overline{K}^{fst})$ and

$T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^{snd}) = \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^{snd})$. It follows that

$$\begin{aligned}
T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}) &= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} (\overline{K}^{fst} \fatsemi \overline{K}^{snd})) \\
&= T((\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^{fst}) \fatsemi (\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^{snd})) \\
&= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^{fst}) \fatsemi T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^{snd}) \\
&= \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^{fst}) \fatsemi \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^{snd}) \\
&= \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} (T(\overline{K}^{fst}) \fatsemi T(\overline{K}^{snd})) \\
&= \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}),
\end{aligned}$$

for all $\overline{K} \in \overline{\Gamma}$. Now assume that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. Then, given a $\overline{K} \in \overline{\Gamma}$, we denote by $\overline{K}^c$ and $\overline{K}^a$ the sub-configurations of $\overline{K}$ whose skeletons are P_c and P_a, respectively. Obviously, $\overline{K} = \langle l_s, \overline{K}|_{l_s} \rangle$`if` C `then` $\overline{K}^c$ `else` $\overline{K}^a$ `endif` $\langle l_f, \overline{K}|_{l_f} \rangle$, for all $\overline{K} \in \overline{\Gamma}$. Moreover, from the induction hypothesis we have that $T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c) = \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^c)$ and $T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a) = \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^a)$. It follows that

$$\begin{aligned}
T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}) &= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \texttt{if } C \texttt{ then } \overline{K}^c \texttt{ else } \overline{K}^a \texttt{ endif } \langle l_f, \overline{K}|_{l_f} \rangle)) \\
&= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} (\langle l_s, \overline{K}|_{l_s} \rangle \texttt{ if } C \texttt{ then } \overline{K}^c \texttt{ else } \overline{K}^a \texttt{ endif } \langle l_f, \overline{K}|_{l_f} \rangle))
\end{aligned}$$

$$= T\left(\begin{array}{l}
\langle l_s, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s} \rangle \\
\quad \texttt{if } C \quad \texttt{then } \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c \\
\qquad\quad \texttt{else } \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a \\
\quad \texttt{endif} \\
\langle l_f, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_f} \rangle
\end{array}\right)$$

$$
\begin{array}{rl}
= & \langle l_s, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \\
& \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \{\sigma \mid \sigma \models C\} \cap \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \fatsemi T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c) \\
& \qquad\qquad \texttt{else } \langle \mathit{first}(P_a), \{\sigma \mid \sigma \models \neg C\} \cap \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \fatsemi T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a) \\
& \quad \texttt{endif} \\
& \langle l_f, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c|_{\mathit{last}(P_c)} \cup \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a|_{\mathit{last}(P_a)}\rangle
\end{array}
$$

$$
\begin{array}{rl}
= & \langle l_s, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \\
& \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \{\sigma \mid \sigma \models C\} \cap \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \fatsemi T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c) \\
& \qquad\qquad \texttt{else } \langle \mathit{first}(P_a), \{\sigma \mid \sigma \models \neg C\} \cap \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \fatsemi T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a) \\
& \quad \texttt{endif} \\
& \langle l_f, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^c|_{\mathit{last}(P_c)} \cup \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^a|_{\mathit{last}(P_a)}\rangle
\end{array}
$$

$$
\begin{array}{rl}
= & (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle l_s, \overline{K}|_{l_s}\rangle) \\
& \quad \texttt{if } C \quad \texttt{then } (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle \mathit{first}(P_c), \{\sigma \mid \sigma \models C\} \cap \overline{K}|_{l_s}\rangle) \fatsemi (\bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^c)) \\
& \qquad\qquad \texttt{else } (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle \mathit{first}(P_a), \{\sigma \mid \sigma \models \neg C\} \cap \overline{K}|_{l_s}\rangle) \fatsemi (\bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^a)) \\
& \quad \texttt{endif} \\
& (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle l_f, \overline{K}^c|_{\mathit{last}(P_c)} \cup \overline{K}^a|_{\mathit{last}(P_a)}\rangle)
\end{array}
$$

$$
= \bigcup_{\overline{K}\in\overline{\Gamma}} \left(\begin{array}{l}
\langle l_s, \overline{K}|_{l_s}\rangle \\
\quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \{\sigma \mid \sigma \models C\} \cap \overline{K}|_{l_s}\rangle \fatsemi T(\overline{K}^c) \\
\qquad\qquad \texttt{else } \langle \mathit{first}(P_a), \{\sigma \mid \sigma \models \neg C\} \cap \overline{K}|_{l_s}\rangle \fatsemi T(\overline{K}^a) \\
\quad \texttt{endif} \\
\langle l_f, \overline{K}^c|_{\mathit{last}(P_c)} \cup \overline{K}^a|_{\mathit{last}(P_a)}\rangle
\end{array} \right)
$$

$$
\begin{aligned}
&= \bigcup_{\overline{K}\in\overline{\Gamma}} T(\langle l_s, \overline{K}|_{l_s}\rangle \texttt{ if } C \texttt{ then } \overline{K}^c \texttt{ else } \overline{K}^a \texttt{ endif } \langle l_f, \overline{K}|_{l_f}\rangle)\\
&= \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}).
\end{aligned}
$$

Finally, assume that P is the `while` program $\langle l_s\rangle$ `while` C `do` P_b `endwhile` $\langle l_f\rangle$. Then, given a configuration $\overline{K} \in \overline{\Gamma}$, we denote by $\overline{K}^b$ the sub-configuration of $\overline{K}$ whose skeleton is P_b, respectively. Obviously, $\overline{K} = \langle l_s, \overline{K}|_{l_s}\rangle$ `while` C `do` $\overline{K}^b$ `endwhile` $\langle l_f, \overline{K}|_{l_f}\rangle$, for all $\overline{K} \in \overline{\Gamma}$. Moreover, from the induction hypothesis we have that $T(\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^b) = \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^b)$. It follows that

$$
\begin{aligned}
T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}) &= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}}(\langle l_s, \overline{K}|_{l_s}\rangle \texttt{ while } C \texttt{ do } \overline{K}^b \texttt{ endwhile } \langle l_f, \overline{K}|_{l_f}\rangle))\\
&= T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}}(\langle l_s, \overline{K}|_{l_s}\rangle \texttt{ while } C \texttt{ do } \overline{K}^b \texttt{ endwhile } \langle l_f, \overline{K}|_{l_f}\rangle))\\
&= T(\langle l_s, \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle \texttt{ while } C \texttt{ then } \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^b \texttt{ endwhile } \langle l_f, \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_f}\rangle)
\end{aligned}
$$

$$
\begin{aligned}
= \quad & \langle l_s, \textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s}\rangle\\
& \texttt{while } C \texttt{ do}\\
& \quad \langle \mathit{first}(P_b), \{\sigma \mid \sigma \models C\} \cap (\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s} \cup \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^b|_{\mathit{last}(P_b)})\rangle\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ;\!\!;T(\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^b)\\
& \texttt{endwhile}\\
& \langle l_f, \{\sigma \mid \sigma \models \neg C\} \cap (\textstyle\bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}|_{l_s} \cup \bigcup_{\overline{K}\in\overline{\Gamma}} \overline{K}^b|_{\mathit{last}(P_b)})\rangle
\end{aligned}
$$

$$
\begin{array}{rl}
= & (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle l_s, \overline{K}|_{l_s}\rangle) \\
& \quad \texttt{while}\ C\ \texttt{do} \\
& \qquad (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle \mathit{first}(P_b), \{\sigma \mid \sigma \models C\} \cap (\overline{K}|_{l_s} \cup \overline{K}^b|_{\mathit{last}(P_b)})\rangle) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \fatsemi(\bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}^b)) \\
& \quad \texttt{endwhile} \\
& (\bigcup_{\overline{K}\in\overline{\Gamma}} \langle l_f, \{\sigma \mid \sigma \models \neg C\} \cap (\overline{K}|_{l_s} \cup \overline{K}^b|_{\mathit{last}(P_b)})\rangle)
\end{array}
$$

$$
= \bigcup_{\overline{K}\in\overline{\Gamma}} \left(\begin{array}{l}
\langle l_s, \overline{K}|_{l_s}\rangle \\
\quad \texttt{while}\ C\ \texttt{do} \\
\qquad \langle \mathit{first}(P_b), \{\sigma \mid \sigma \models C\} \cap (\overline{K}|_{l_s} \cup \overline{K}^b|_{\mathit{last}(P_b)})\rangle \fatsemi T(\overline{K}^b) \\
\quad \texttt{endwhile} \\
\langle l_f, \{\sigma \mid \sigma \models \neg C\} \cap (\overline{K}|_{l_s} \cup \overline{K}^b|_{\mathit{last}(P_b)})\rangle
\end{array} \right)
$$

$$
\begin{array}{rl}
= & \bigcup_{\overline{K}\in\overline{\Gamma}} T(\langle l_s, \overline{K}|_{l_s}\rangle\ \texttt{while}\ C\ \texttt{do}\ \overline{K}^b\ \texttt{endwhile}\ \langle l_f, \overline{K}|_{l_f}\rangle) \\
= & \bigcup_{\overline{K}\in\overline{\Gamma}} T(\overline{K}).
\end{array}
$$

which completes the proof by induction. □

10.5 Proposition We have that $T^p \circ \vec{\alpha} = \vec{\alpha} \circ \vec{T}$.

Proof: We will show that given a program P, $(T^p \circ \vec{\alpha})(\vec{\Gamma}) = (\vec{\alpha} \circ \vec{T})(\vec{\Gamma})$, for all $\vec{\Gamma} \subseteq \{\vec{K} \mid |\vec{K}| = P\}$. The proof is by induction on the structure of program P. For the purpose of this proof, we denote by $\lfloor\{a\}\rfloor$. Let $\vec{\Gamma}$ be a set of singleton configurations whose skeleton is P. First, assume that P is the skip program $\langle l_s\rangle\ \texttt{skip}\ \langle l_f\rangle$. We can express $\vec{\Gamma}$ as $\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\ \texttt{skip}\ \langle l_f, \vec{K}|_{l_f}\rangle\}$. Then, using Remarks 10.3 and

10.4 we have that

$$T^p(\vec{\alpha}(\vec{\Gamma})) = T^p(\langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle \,\mathtt{skip}\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_f}\rangle) = \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle \,\mathtt{skip}\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle.$$

We also have that

$$\vec{\alpha}(\vec{T}(\vec{\Gamma})) = \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}} \{\langle l_s, \vec{K}|_{l_s}\rangle \,\mathtt{skip}\, \langle l_f, \vec{K}|_{l_s}\rangle\}) = \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle \,\mathtt{skip}\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle.$$

It follows that $T^p(\vec{\alpha}(\vec{\Gamma})) = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. Next, assume that P is the assignment $\langle l_s\rangle\, x := E\, \langle l_f\rangle$. Then, we can express the set $\vec{\Gamma}$ as $\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\, x := E\, \langle l_f, \vec{K}|_{l_f}\rangle\}$. We have that

$$\begin{array}{rcl} T^p(\vec{\alpha}(\vec{\Gamma})) & = & T^p(\langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle\, x := E\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_f}\rangle) \\ & = & \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle\, x := E\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}[x \mapsto E]\rangle. \end{array}$$

We also have

$$\begin{array}{rcl} \vec{\alpha}(\vec{T}(\vec{\Gamma})) & = & \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\, x := E\, \langle l_f, \vec{K}|_{l_f}[x \mapsto E]\rangle\}) \\ & = & \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle\, x := E\, \langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}[x \mapsto E]\rangle. \end{array}$$

It follows that $T^p(\vec{\alpha}(\vec{\Gamma})) = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. Assume now that P is the sequence program $P_1 \fatsemi P_2$. Given a configuration $\vec{K} \in \vec{\Gamma}$, we denote by $\vec{K}^{fst}$ and $\vec{K}^{snd}$ the sub-configurations of $\vec{K}$ whose skeletons are P_1 and P_2, respectively. We can express the set $\vec{\Gamma}$ as $\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst} \fatsemi \vec{K}^{snd}\}$. Then, using Remarks 10.3 and 10.4, we have that

$$\begin{array}{rcl} T^p(\vec{\alpha}(\vec{\Gamma})) & = & T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\}) \fatsemi \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\})) \\ & = & T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\})) \fatsemi T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\})). \end{array}$$

We also have

$$\begin{array}{rcl} \vec{\alpha}(\vec{T}(\vec{\Gamma})) & = & \vec{\alpha}(\bigcup_{\vec{K}\in\vec{T}(\vec{\Gamma})})\{\vec{K}^{fst} \fatsemi \vec{K}^{snd}\} \\ & = & \bigcup_{\vec{K}\in\vec{T}(\vec{\Gamma})} \vec{\alpha}(\{\vec{K}^{fst}\}) \fatsemi \vec{\alpha}(\{\vec{K}^{snd}\}) \\ & = & \vec{\alpha}(\bigcup_{\vec{K}\in\vec{T}(\vec{\Gamma})}\{\vec{K}^{fst}\}) \fatsemi \vec{\alpha}(\bigcup_{\vec{K}\in\vec{T}(\vec{\Gamma})}\{\vec{K}^{snd}\}) \\ & = & \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\})) \fatsemi \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\})). \end{array}$$

From the induction hypothesis, we have that $T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\}) = \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{fst}\}))$ and $T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\}) = \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^{snd}\}))$, from which it follows that $T^p(\vec{\alpha}(\vec{\Gamma})) = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. Assume now that P is the `if` program $\langle l_s\rangle\,\texttt{if}\ C\ \texttt{then}\ P_c\ \texttt{else}\ P_a\ \texttt{endif}\,\langle l_f\rangle$. Given a configuration $\vec{K} \in \vec{\Gamma}$, we denote by $\vec{K}^c$ and $\vec{K}^a$ the sub-configurations of $\vec{K}$ whose skeletons are P_c and P_a, respectively. Then, we can write the set $\vec{\Gamma}$ as $\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle\,\texttt{if}\ C\ \texttt{then}\ \vec{K}^c\ \texttt{else}\ \vec{K}^a\ \texttt{endif}\,\langle l_f, \vec{K}|_{l_f}\rangle\}$. Then, using Remarks 10.3 and 10.4, we have that

$$
\begin{array}{rcl}
T^p(\vec{\alpha}(\vec{\Gamma})) & = & T^p\left(\begin{array}{l}
\langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s}\rangle \\
\quad \texttt{if}\ C\ \texttt{then}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\}) \\
\quad \texttt{else}\ \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\}) \\
\quad \texttt{endif}\,\langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_f}\rangle
\end{array}\right) \\
& = & \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s}\rangle \\
& & \quad \texttt{if}\ C \quad \texttt{then}\ \langle \mathit{first}(P_c), \mathit{filter}(C, \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\})) \\
& & \qquad\qquad \texttt{else}\ \langle \mathit{first}(P_a), \mathit{filter}(\neg C, \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\})) \\
& & \quad \texttt{endif} \\
& & \langle l_f, (\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}^c|_{\mathit{last}(P_c)}) \cup (\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}^a|_{\mathit{last}(P_a)})\rangle.
\end{array}
$$

We also have that

$$
\begin{array}{rcl}
\vec{\alpha}(\vec{T}(\vec{\Gamma})) & = & \vec{\alpha}\left(\bigcup_{\vec{K}\in\vec{\Gamma}}\left\{\begin{array}{l}\langle l_s,\vec{K}|_{l_s}\rangle\\ \quad \texttt{if } C \quad \texttt{then}\langle \mathit{first}(P_c),\mathit{filter}(C,\vec{K}|_{l_s})\rangle \fatsemi \vec{K}_1\\ \qquad\quad \texttt{else } \langle \mathit{first}(P_a),\mathit{filter}(\neg C,\vec{K}|_{l_s})\rangle \fatsemi \vec{K}_2\\ \quad \texttt{endif}\\ \langle l_f,\vec{K}^c|_{\mathit{last}(P_c)}\cup\vec{K}^a|_{\mathit{last}(P_a)}\rangle \mid \vec{K}_1\in\vec{T}(\{\vec{K}^c\}),\vec{K}_2\in\vec{T}(\{\vec{K}^a\})\end{array}\right\}\right)\\
& = & \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{\alpha}\left(\left\{\begin{array}{l}\langle l_s,\vec{K}|_{l_s}\rangle\\ \quad \texttt{if } C \quad \texttt{then}\langle \mathit{first}(P_c),\mathit{filter}(C,\vec{K}|_{l_s})\rangle \fatsemi \vec{K}_1\\ \qquad\quad \texttt{else } \langle \mathit{first}(P_a),\mathit{filter}(\neg C,\vec{K}|_{l_s})\rangle \fatsemi \vec{K}_2\\ \quad \texttt{endif}\\ \langle l_f,\vec{K}^c|_{\mathit{last}(P_c)}\cup\vec{K}^a|_{\mathit{last}(P_a)}\rangle \mid \vec{K}_1\in\vec{T}(\{\vec{K}^c\}),\vec{K}_2\in\vec{T}(\{\vec{K}^a\})\end{array}\right\}\right)\\
& = & \bigcup_{\vec{K}\in\vec{\Gamma}}\left(\begin{array}{l}\langle l_s,\vec{K}|_{l_s}\rangle\\ \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c),\mathit{filter}(C,\vec{K}|_{l_s})\rangle \fatsemi \vec{\alpha}(\vec{T}(\{\vec{K}^c\}))\\ \qquad\quad \texttt{else } \langle \mathit{first}(P_a),\mathit{filter}(\neg C,\vec{K}|_{l_s})\rangle \fatsemi \vec{\alpha}(\vec{T}(\{\vec{K}^a\}))\\ \quad \texttt{endif}\\ \langle l_f,\vec{K}^c|_{\mathit{last}(P_c)}\cup\vec{K}^a|_{\mathit{last}(P_a)}\rangle\end{array}\right)\\
& = & \langle l_s,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s}\rangle\\
& & \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c),\mathit{filter}(C,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{\alpha}(\vec{T}(\{\vec{K}^c\}))\\
& & \qquad\quad \texttt{else } \langle \mathit{first}(P_a),\mathit{filter}(\neg C,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi \bigcup_{\vec{K}\in\vec{\Gamma}}\vec{\alpha}(\vec{T}(\{\vec{K}^a\}))\\
& & \quad \texttt{endif}\\
& & \langle l_f,\bigcup_{\vec{K}\in\vec{\Gamma}}(\vec{K}^c|_{\mathit{last}(P_c)}\cup\vec{K}^a|_{\mathit{last}(P_a)})\rangle\\
& = & \langle l_s,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s}\rangle\\
& & \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c),\mathit{filter}(C,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\}))\\
& & \qquad\quad \texttt{else } \langle \mathit{first}(P_a),\mathit{filter}(\neg C,\bigcup_{\vec{K}\in\vec{\Gamma}}\vec{K}|_{l_s})\rangle \fatsemi \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\}))\\
& & \quad \texttt{endif}\\
& & \langle l_f,\bigcup_{\vec{K}\in\vec{\Gamma}}(\vec{K}^c|_{\mathit{last}(P_c)}\cup\vec{K}^a|_{\mathit{last}(P_a)})\rangle.
\end{array}
$$

Since from the induction hypothesis we have that $T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\})) = \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^c\}))$ and $T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\})) = \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^a\}))$, it follows that $T^p(\vec{\alpha}(\vec{\Gamma})) = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. Finally, assume that P is the `while` program $\langle l_s\rangle$ `while` C `do` P_b `endwhile` $\langle l_f\rangle$. Given a configuration $\vec{K} \in \vec{\Gamma}$, we denote by $\vec{K}^b$ the sub-configuration of $\vec{K}$ whose skeleton is P_b. Then, we can write the set $\vec{\Gamma}$ as $\bigcup_{\vec{K}\in\vec{\Gamma}}\{\langle l_s, \vec{K}|_{l_s}\rangle$ `while` C `do` $\vec{K}^b$ `endwhile` $\langle l_f, \vec{K}|_{l_f}\rangle\}$. Then, using Remarks 10.3 and 10.4, we have that

$$
\begin{array}{rcl}
T^p(\vec{\alpha}(\vec{\Gamma})) & = & T^p\left(\begin{array}{l}
\langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle \\
\quad \texttt{while } C \texttt{ do} \\
\qquad \vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^b\}) \\
\quad \texttt{endwhile} \\
\langle l_f, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_f}\rangle
\end{array}\right) \\
& = & \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \qquad \langle \mathit{first}(P_c), \mathit{filter}(C, \mathit{before}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \\
& & \qquad\qquad \fatsemi T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^b\})) \\
& & \quad \texttt{endwhile} \\
& & \langle l_f, \mathit{filter}(\neg C, (\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}) \cup \mathit{collect}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle.
\end{array}
$$

We also have that

$$
\vec{\alpha}(\vec{T}(\vec{\Gamma})) = \vec{\alpha}\left(\bigcup_{\vec{K}\in\vec{\Gamma}}\left\{\begin{array}{l}
\langle l_s, \vec{K}|_{l_s}\rangle \\
\quad \texttt{while } C \texttt{ do} \\
\qquad \langle \mathit{first}(P_b), \mathit{filter}(C, \mathit{before}(\vec{K}|_{l_s}, \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \fatsemi \vec{K}_1 \\
\quad \texttt{endwhile} \\
\langle l_f, \mathit{filter}(\neg C, \vec{K}|_{l_s} \cup \mathit{collect}(\vec{K}^b|_{\mathit{last}(P_b)}))\rangle \mid \vec{K}_1 \in \vec{T}(\{\vec{K}^b\})
\end{array}\right\}\right)
$$

$$= \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{\alpha} \left(\left\{ \begin{array}{l} \langle l_s, \vec{K}|_{l_s}\rangle \\ \quad \texttt{while } C \texttt{ do} \\ \quad\quad \langle \mathit{first}(P_b), \mathit{filter}(C, \mathit{before}(\vec{K}|_{l_s}, \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \fatsemi \vec{K}_1 \\ \quad \texttt{endwhile} \\ \langle l_f, \mathit{filter}(\neg C, \vec{K}|_{l_s} \cup \mathit{collect}(\vec{K}^b|_{\mathit{last}(P_b)}))\rangle \mid \vec{K}_1 \in \vec{T}(\{\vec{K}^b\}) \end{array} \right\} \right)$$

$$= \bigcup_{\vec{K}\in\vec{\Gamma}} \left(\begin{array}{l} \langle l_s, \vec{K}|_{l_s}\rangle \\ \quad \texttt{while } C \texttt{ do} \\ \quad\quad \langle \mathit{first}(P_b), \mathit{filter}(C, \mathit{before}(\vec{K}|_{l_s}, \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \fatsemi \vec{\alpha}(\vec{T}(\{\vec{K}^b\})) \\ \quad \texttt{endwhile} \\ \langle l_f, \mathit{filter}(\neg C, \vec{K}|_{l_s} \cup \mathit{collect}(\vec{K}^b|_{\mathit{last}(P_b)}))\rangle \end{array} \right)$$

$$= \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle$$
$$\texttt{while } C \texttt{ do}$$
$$\langle \mathit{first}(P_b), \mathit{filter}(C, \mathit{before}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \fatsemi \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{\alpha}(\vec{T}(\{\vec{K}^b\}))$$
$$\texttt{endwhile}$$
$$\langle l_f, \mathit{filter}(\neg C, (\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}) \cup \mathit{collect}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle$$

$$= \langle l_s, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}\rangle$$
$$\texttt{while } C \texttt{ do}$$
$$\langle \mathit{first}(P_b), \mathit{filter}(C, \mathit{before}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}, \bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle \fatsemi \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}} \{\vec{K}^b\}))$$
$$\texttt{endwhile}$$
$$\langle l_f, \mathit{filter}(\neg C, (\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}|_{l_s}) \cup \mathit{collect}(\bigcup_{\vec{K}\in\vec{\Gamma}} \vec{K}^b|_{\mathit{last}(P_b)}))\rangle.$$

Since from the induction hypothesis we have that $T^p(\vec{\alpha}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^b\})) = \vec{\alpha}(\vec{T}(\bigcup_{\vec{K}\in\vec{\Gamma}}\{\vec{K}^b\}))$, it follows that $T^p(\vec{\alpha}(\vec{\Gamma})) = \vec{\alpha}(\vec{T}(\vec{\Gamma}))$. $\square$

10.11 Proposition Given a program P, we have that $T \circ \overline{\alpha} = \overline{\alpha} \circ T^p$.

Proof: We need to show that $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$, for all $K \in \mathbf{Progressive}(K)$. The proof is by induction on the structure of the program P. Assume first that P is the skip program $\langle l_s\rangle\,\mathtt{skip}\,\langle l_f\rangle$. Then, K can be written as $\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle\,\mathtt{skip}\,\langle \mathit{last}(P), K|_{\mathit{last}(P)}\rangle$. We have that

$$\begin{aligned} T(\overline{\alpha}(K)) &= T(\overline{\alpha}(\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle)\,\mathtt{skip}\,\overline{\alpha}(\langle \mathit{last}(P), K|_{\mathit{last}(P)}\rangle)) \\ &= T(\langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\,\mathtt{skip}\,\langle \mathit{last}(P), \bigcup_{\tilde{\mu}} K|_{\mathit{last}(P)}(\tilde{\mu})\rangle) \\ &= \langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\ \mathtt{skip}\ \langle \mathit{last}(P), \bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle \end{aligned}$$

We also have that

$$\begin{aligned} \overline{\alpha}(T^p(K)) &= \overline{\alpha}(\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle\ \mathtt{skip}\ \langle \mathit{last}(P), K|_{\mathit{first}(P)}\rangle) \\ &= \langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\ \mathtt{skip}\ \langle \mathit{last}(P), \bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle. \end{aligned}$$

It follows that $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$. Assume now that P is the assignment $\langle l_s\rangle\, x := E\,\langle l_f\rangle$. Then, K can be written as $\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle\, x := E\,\langle \mathit{last}(P), K|_{\mathit{last}(P)}\rangle$. We have that

$$\begin{aligned} T(\overline{\alpha}(K)) &= T(\overline{\alpha}(\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle)\ x := E\ \overline{\alpha}(\langle \mathit{last}(P), K|_{\mathit{last}(P)}\rangle)) \\ &= T(\langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\ x := E\ \langle \mathit{last}(P), \bigcup_{\tilde{\mu}} K|_{\mathit{last}(P)}(\tilde{\mu})\rangle) \\ &= T(\langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\, x := E\,\langle \mathit{last}(P), (\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu}))[x \mapsto E]\rangle). \end{aligned}$$

We also have that

$$\begin{aligned} \overline{\alpha}(T^p(K)) &= \overline{\alpha}(\langle \mathit{first}(P), K|_{\mathit{first}(P)}\rangle\, x := E\,\langle \mathit{last}(P), \widehat{\mathit{assign}}(K|_{\mathit{first}(P)}, x, E)\rangle) \\ &= T(\langle \mathit{first}(P), \textstyle\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu})\rangle\, x := E\,\langle \mathit{last}(P), (\bigcup_{\tilde{\mu}} K|_{\mathit{first}(P)}(\tilde{\mu}))[x \mapsto E]\rangle). \end{aligned}$$

It follows that $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$. Next, assume that P is the sequence program $P_1 \fatsemi P_2$. Given a configuration K whose skeleton is P, denote by K^{fst} and K^{snd} the

sub-configurations of K whose skeleton are P_1 and P_2, respectively. Then, K can be written as $K^{fst} \fatsemi K^{snd}$. We have that

$$\begin{aligned} T(\overline{\alpha}(K)) &= T(\overline{\alpha}(K^{fst} \fatsemi K^{snd})) \\ &= T(\overline{\alpha}(K^{fst})) \fatsemi T(\overline{\alpha}(K^{snd})). \end{aligned}$$

We also have that

$$\begin{aligned} \overline{\alpha}(T^p(K)) &= \overline{\alpha}(T^p(K^{fst} \fatsemi K^{snd})) \\ &= \overline{\alpha}(T^p(K^{fst})) \fatsemi \overline{\alpha}(T^p(K^{snd})). \end{aligned}$$

From the induction hypothesis we have that $T(\overline{\alpha}(K^{fst})) = \overline{\alpha}(T^p(K^{fst}))$ and $T(\overline{\alpha}(K^{snd})) = \overline{\alpha}(T^p(K^{snd}))$, from which it follows that $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$. Next, assume that P is the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$. Given a configuration K whose skeleton is P, denote by K^c and K^a the sub-configurations of K whose skeleton are P_c and P_a, respectively. Then, K can be written as $\langle l_s, K|_{l_s} \rangle$ `if` C `then` K^c `else` K^a `endif` $\langle l_f, K|_{l_f} \rangle$. We have that

$$\begin{aligned} T(\overline{\alpha}(K)) &= T(\overline{\alpha}(\langle l_s, K|_{l_s} \rangle \texttt{ if } C \texttt{ then } K^c \texttt{ else } K^a \texttt{ endif } \langle l_f, K|_{l_f} \rangle)) \\ &= T(\langle l_s, \textstyle\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \rangle \texttt{ if } C \texttt{ then } \overline{\alpha}(K^c) \texttt{ else } \overline{\alpha}(K^a) \texttt{ endif } \langle l_f, \textstyle\bigcup_{\tilde{\mu}} K|_{l_f}(\tilde{\mu}) \rangle) \\ &= \langle l_s, \textstyle\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \rangle \\ &\qquad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \{\sigma \mid C \models \sigma\} \cap \textstyle\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \rangle \fatsemi T(\overline{\alpha}(K^c)) \\ &\qquad\qquad \texttt{else } \langle \mathit{first}(P_c), \{\sigma \mid \neg C \models \sigma\} \cap \textstyle\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \rangle \fatsemi T(\overline{\alpha}(K^a)) \\ &\qquad \texttt{endif} \quad \langle l_f, \overline{\alpha}(K^c|_{\mathit{last}(P_c)}) \cup \overline{\alpha}(K^a|_{\mathit{last}(P_a)}) \rangle. \end{aligned}$$

We also have that

$$\overline{\alpha}(T^p(K)) = \overline{\alpha}\left(\begin{array}{l} \langle l_s, K|_{l_s}\rangle \\ \quad \texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \mathit{filter}(C, K|_{l_s})\rangle \fatsemi T^p(K^c) \\ \qquad\quad \texttt{else } \langle \mathit{first}(P_a), \mathit{filter}(\neg C, K|_{l_s})\rangle \fatsemi T^p(K^a) \\ \quad \texttt{endif} \\ \langle l_f, K^c|_{\mathit{last}(P_c)} \cup K^a|_{\mathit{last}(P_a)}\rangle \end{array}\right)$$

$$= \langle l_s, \textstyle\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu})\rangle$$

$$\texttt{if } C \quad \texttt{then } \langle \mathit{first}(P_c), \textstyle\bigcup_{\tilde{\mu}}(\mathit{filter}(C, K|_{l_s}))(\tilde{\mu})\rangle \fatsemi \overline{\alpha}(T^p(K^c))$$

$$\texttt{else } \langle \mathit{first}(P_a), \textstyle\bigcup_{\tilde{\mu}}(\mathit{filter}(\neg C, K|_{l_s}))(\tilde{\mu})\rangle \fatsemi \overline{\alpha}(T^p(K^a))$$

$$\texttt{endif} \quad \langle l_f, \textstyle\bigcup_{\tilde{\mu}}(K^c|_{\mathit{last}(P_c)} \cup K^a|_{\mathit{last}(P_a)})(\tilde{\mu})\rangle.$$

From the induction hypothesis we have $\overline{\alpha}(T^p(K^c)) = T(\overline{\alpha}(K^c))$ and $\overline{\alpha}(T^p(K^a)) = T(\overline{\alpha}(K^a))$, and using Remarks 10.9 and 10.10 we obtain $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$. Finally, assume that P is the `while` program $\langle l_s\rangle$ `while` C `do` P_b `endwhile` $\langle l_f\rangle$. Given a configuration K whose skeleton is P, denote by K^b the sub-configuration of K whose skeleton is P_b. Then, K can be written as

$\langle l_s, K|_{l_s}\rangle$ `while` C `do` K^b `endwhile` $\langle l_f, K|_{l_f}\rangle$. We have that

$$
\begin{array}{rcl}
T(\overline{\alpha}(K)) & = & T(\overline{\alpha}(\langle l_s, K|_{l_s}\rangle \texttt{ while } C \texttt{ do } K^b \texttt{ endwhile } \langle l_f, K|_{l_f}\rangle)) \\
& = & T(\langle l_s, \bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu})\rangle \texttt{ while } C \texttt{ do } \overline{\alpha}(K^b) \texttt{ endwhile } \langle l_f, \bigcup_{\tilde{\mu}} K|_{l_f}(\tilde{\mu})\rangle) \\
& = & \langle l_s, \bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu})\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \quad\quad \langle \mathit{first}(P_b), \{\sigma \mid \sigma \models C\} \cap (\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \cup \overline{\alpha}(K^b)|_{\mathit{last}(P_b)})\rangle \fatsemi T(\overline{\alpha}(K^b)) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle \mathit{last}(P_b), \{\sigma \mid \sigma \models \neg C\} \cap (\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \cup \overline{\alpha}(K^b)|_{\mathit{last}(P_b)})\rangle \\
& = & \langle l_s, \bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu})\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \quad\quad \langle \mathit{first}(P_b), \{\sigma \mid \sigma \models C\} \cap (\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \cup \bigcup_{\tilde{\mu}} K^b|_{\mathit{last}(P_b)}(\tilde{\mu}))\rangle \fatsemi T(\overline{\alpha}(K^b)) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle \mathit{last}(P_b), \{\sigma \mid \sigma \models \neg C\} \cap (\bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu}) \cup \bigcup_{\tilde{\mu}} K^b|_{\mathit{last}(P_b)}(\tilde{\mu}))\rangle.
\end{array}
$$

We also have that

$$
\begin{array}{rcl}
\overline{\alpha}(T^p(K)) & = & \overline{\alpha}\left(\begin{array}{l}
\langle l_s, K|_{l_s}\rangle \\
\quad \texttt{while } C \texttt{ then} \\
\quad\quad \langle \mathit{first}(P_c), \mathit{filter}(C, \mathit{before}(K|_{l_s}, K^b|_{\mathit{last}(P_b)}))\rangle \fatsemi T^p(K^b) \\
\quad \texttt{endwhile} \\
\langle l_f, \mathit{filter}(\neg C, \mathit{collect}(K^c|_{\mathit{last}(P_c)}) \cup K^a|_{\mathit{last}(P_a)})\rangle
\end{array}\right) \\
& = & \langle l_s, \bigcup_{\tilde{\mu}} K|_{l_s}(\tilde{\mu})\rangle \\
& & \quad \texttt{while } C \texttt{ then} \\
& & \quad\quad \langle \mathit{first}(P_c), \bigcup_{\tilde{\mu}} (\mathit{filter}(C, \mathit{before}(K|_{l_s}, K^b|_{\mathit{last}(P_b)})))(\tilde{\mu})\rangle \fatsemi \overline{\alpha}(T^p(K^b)) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle l_f, \bigcup_{\tilde{\mu}} (\mathit{filter}(\neg C, \mathit{collect}(K^c|_{\mathit{last}(P_c)}) \cup K^a|_{\mathit{last}(P_a)}))(\tilde{\mu})\rangle
\end{array}
$$

From the induction hypothesis we have $\overline{\alpha}(T^p(K^b)) = T(\overline{\alpha}(K^b))$, and using Remarks 10.9 and 10.10 it follows that $T(\overline{\alpha}(K)) = \overline{\alpha}(T^p(K))$. $\square$

11.3 Proposition Let K be a progressive configuration, and $\hat{K}$ a family configuration that covers K. Then, $\widehat{T}(\hat{K})$ covers $T^p(K)$.

Proof: The proof is by induction on the structure of the configuration K. Assume first that K is the `skip` statement $\langle l_s, \Psi_s\rangle\,\texttt{skip}\,\langle l_f, \Psi_f\rangle$. Since $\hat{K}$ covers K, it must be of the form $\langle l_s, \Phi_s\rangle\,\texttt{skip}\,\langle l_f, \Phi_f\rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively. $T^p(K)$ is the progressive configuration $\langle l_s, \Psi_s\rangle\,\texttt{skip}\,\langle l_f, \Psi_s\rangle$, and $\widehat{T}(\hat{K})$ is the family configuration $\langle l_s, \Phi_s\rangle\,\texttt{skip}\,\langle l_f, \Phi_s\rangle$. Clearly, $\widehat{T}(\hat{K})$ covers $T^p(K)$. Assume now that K is the assignment $\langle l_s, \Psi_s\rangle\, x := E\,\langle l_f, \Psi_f\rangle$. Since $\hat{K}$ covers K, it must be of the form $\langle l_s, \Phi_s\rangle\, x := E\,\langle l_f, \Phi_f\rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively. $T^p(K)$ is the progressive configuration $\langle l_s, \Psi_s\rangle\, x := E\,\langle l_f, \mathit{assign}(x, E, \Psi_f)\rangle$, and $\widehat{T}(\hat{K})$ is the family configuration $\langle l_s, \Phi_s\rangle\, x := E\,\langle l_f, \widehat{\mathit{assign}}(x, E, \Phi_f)\rangle$. According to Remark 11.2, $\widehat{T}(\hat{K})$ covers $T^p(K)$. Next, assume that K is the sequence $K_1 \fatsemi K_2$. According to Remark 11.1 $\hat{K}$ is a family configuration $\hat{K}_1 \fatsemi \hat{K}_2$, where $\hat{K}_1$ covers K_1 and $\hat{K}_2$ covers K_2. Using the induction hypothesis, we have that $\widehat{T}(\hat{K}_1)$ covers $T^p(K_1)$ and $\widehat{T}(\hat{K}_2)$ covers $T^p(K_2)$. Using Remark 11.1 again, we have that $\widehat{T}(\hat{K}_1) \fatsemi \widehat{T}(\hat{K}_2)$ covers $T^p(K_1) \fatsemi T^p(K_2)$, which entails that $\widehat{T}(\hat{K})$ covers $T^p(K)$. Assume now that K is the `if` statement $\langle l_s, \Psi_s\rangle\,\texttt{if}\ C\ \texttt{then}\ K_c\ \texttt{else}\ K_a\ \texttt{endif}\,\langle l_f, \Psi_f\rangle$. According to Remark 11.1 $\hat{K}$ is a family configuration $\langle l_s, \Phi_s\rangle\,\texttt{if}\ C\ \texttt{then}\ \hat{K}_c\ \texttt{else}\ \hat{K}_a\ \texttt{endif}\,\langle l_f, \Phi_f\rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively, and $\hat{K}_c$ and $\hat{K}_a$ covering K_c and K_a, respectively. Then, $T^p(K)$ is the progressive configuration $\langle l_s, \Psi_s\rangle\,\texttt{if}\ C\ \texttt{then}\ \langle \mathit{first}(K_c), \mathit{filter}(C, \Psi_s)\rangle \fatsemi T^p(K_c)\ \texttt{else}\ \langle \mathit{first}(K_a), \mathit{filter}(\neg C, \Psi_s)\rangle \fatsemi T^p(K_a)\ \ \texttt{endif}\,\langle l_f, K_c|_{\mathit{last}(K_c)} \cup K_a|_{\mathit{last}(K_a)}\rangle$. On the other hand, $\widehat{T}(\hat{K})$ is the family configuration $\langle l_s, \Phi_s\rangle\,\texttt{if}\ \ C\ \ \texttt{then}\ \ \langle \mathit{first}(\hat{K}_c), \widehat{\mathit{filter}}(C, \Phi_s)\rangle \fatsemi \widehat{T}(\hat{K}_c)\ \texttt{else}\ \langle \mathit{first}(\hat{K}_a), \widehat{\mathit{filter}}(\neg C, \Phi_s)\rangle \fatsemi \widehat{T}(\hat{K}_a)\ \texttt{endif}\,\langle l_f, \hat{K}_c|_{\mathit{last}(\hat{K}_c)} \widehat{\cup} \hat{K}_a|_{\mathit{last}(\hat{K}_a)}\rangle$.

Using the induction hypothesis, we have that $\widehat{T}(\hat{K}_c)$ covers $T^p(K_c)$ and $\widehat{T}(\hat{K}_a)$ covers $T^p(K_a)$. Using Remark 11.2, we also have that $\widehat{\mathit{filter}}(C, \Phi_s)$ covers $\mathit{filter}(C, \Psi_s)$, $\widehat{\mathit{filter}}(\neg C, \Phi_s)$ covers $\mathit{filter}(\neg C, \Psi_s)$, and $\hat{K}_c|_{\mathit{last}(\hat{K}_c)} \widehat{\cup} \hat{K}_a|_{\mathit{last}(\hat{K}_a)}$ covers $K_c|_{\mathit{last}(K_c)} \cup K_a|_{\mathit{last}(K_a)}$. Using Remark 11.1 again, it follows that $\widehat{T}(\hat{K})$ covers $T^p(K)$. Finally, assume that K is the `while` statement $\langle l_s, \Psi_s\rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_f\rangle$. According to Remark 11.1 $\hat{K}$ is a family configuration $\langle l_s, \Phi_s\rangle$ `while` C `do` $\hat{K}_b$ `endwhile` $\langle l_f, \Phi_f\rangle$, with Φ_s and Φ_f covering Ψ_s and Ψ_f, respectively, and $\hat{K}_b$ covering K_b. Then, $T^p(K)$ is the progressive configuration $\langle l_s, \Psi_s\rangle$ `while` C `do` $\langle \mathit{first}(K_b), \mathit{filter}(C, \mathit{before}(\Psi_s, K_b|_{\mathit{last}(K_b)}))\rangle$ ⨟ $T^p(K_c)$ `endwhile` $\langle l_f, \mathit{filter}(\neg C, \Psi_s \cup \mathit{collect}K_b|_{\mathit{last}(K_b)})\rangle$. On the other hand, $\widehat{T}(\hat{K})$ is the family configuration $\langle l_s, \Phi_s\rangle$ `while` C `do` $\langle \mathit{first}(\hat{K}_b), \widehat{\mathit{filter}}(C, \widehat{\mathit{before}}(\Phi_s, \hat{K}_b|_{\mathit{last}(\hat{K}_b)}))\rangle$ ⨟ $\widehat{T}(\hat{K}_c)$ `endwhile` $\langle l_f, \mathit{filter}(\neg C, \Phi_s \cup \mathit{collect}\hat{K}_b|_{\mathit{last}(\hat{K}_b)})\rangle$. Using the induction hypothesis, we have that $\widehat{T}(\hat{K}_b)$ covers $T^p(K_b)$. Using Remark 11.2, we also have that $\widehat{\mathit{filter}}(C, \widehat{\mathit{before}}(\Phi_s, \hat{K}_b|_{\mathit{last}(\hat{K}_b)}))$ covers $\mathit{filter}(C, \mathit{before}(\Psi_s, K_b|_{\mathit{last}(K_b)}))$ and $\mathit{filter}(\neg C, \Phi_s \cup \mathit{collect}\hat{K}_b|_{\mathit{last}(\hat{K}_b)})$ covers $\mathit{filter}(\neg C, \Psi_s \cup \mathit{collect}K_b|_{\mathit{last}(K_b)})$. Using Remark 11.1 again, it follows that $\widehat{T}(\hat{K})$ covers $T^p(K)$. □

11.4 Proposition Let P be the sequence program $P_1 ⨟ P_2$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_1$ and $\hat{K}_2$ the sub-configurations of $\vec{K}$ such that $|\hat{K}_1| = P_1$ and $|\hat{K}_2| = P_2$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_1) \preccurlyeq \hat{K}_1$ and $\widehat{T}(\hat{K}_2) \preccurlyeq \hat{K}_2$.

Proof: We have that $|\widehat{T}(\hat{K})| = |\hat{K}| = P$, which entails that $\mathit{labels}(\widehat{T}(\hat{K})) = \mathit{labels}(\hat{K}) = \mathit{labels}(P)$. Similarly, we have $\mathit{labels}(\widehat{T}(\hat{K}_1)) = \mathit{labels}(\hat{K}_1) = \mathit{labels}(P_1)$ and $\mathit{labels}(\widehat{T}(\hat{K}_2)) = \mathit{labels}(\hat{K}_2) = \mathit{labels}(P_2)$. Since P_1 and P_2 are subprograms of P, if follows that $\mathit{labels}(P_1) \subseteq \mathit{labels}(P)$ and $\mathit{labels}(P_2) \subseteq \mathit{labels}(P)$. Now, $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$

entails that $\widehat{T}(\hat{K})|_l \preccurlyeq \hat{K}|_l$, for all $l \in \mathit{labels}(P)$. Since $\mathit{labels}(P_1) \subseteq \mathit{labels}(P)$ and $\mathit{labels}(P_2) \subseteq \mathit{labels}(P)$, we have that $\widehat{T}(\hat{K}_1) \preccurlyeq \hat{K}_1$ and $\widehat{T}(\hat{K}_2) \preccurlyeq \hat{K}_2$. $\square$

11.5 Proposition Let P be the `if` program $\langle l_s \rangle$ `if` C `then` P_c `else` P_a `endif` $\langle l_f \rangle$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_c$ and $\hat{K}_a$ the sub-configurations of $\hat{K}$ such that $|\hat{K}_a| = P_a$ and $|\hat{K}_c| = P_c$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_a) \preccurlyeq \hat{K}_a$ and $\widehat{T}(\hat{K}_c) \preccurlyeq \hat{K}_c$.

Proof: We have that $|\widehat{T}(\hat{K})| = |\hat{K}| = P$, which entails that $\mathit{labels}(\widehat{T}(\hat{K})) = \mathit{labels}(\hat{K}) = \mathit{labels}(P)$. Similarly, we have $\mathit{labels}(\widehat{T}(\hat{K}_c)) = \mathit{labels}(\hat{K}_c) = \mathit{labels}(P_c)$ and $\mathit{labels}(\widehat{T}(\hat{K}_a)) = \mathit{labels}(\hat{K}_a) = \mathit{labels}(P_a)$. Since P_c and P_a are subprograms of P, if follows that $\mathit{labels}(P_c) \subseteq \mathit{labels}(P)$ and $\mathit{labels}(P_a) \subseteq \mathit{labels}(P)$. Now, $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$ entails that $\widehat{T}(\hat{K})|_l \preccurlyeq \hat{K}|_l$, for all $l \in \mathit{labels}(P)$. Since $\mathit{labels}(P_c) \subseteq \mathit{labels}(P)$ and $\mathit{labels}(P_c) \subseteq \mathit{labels}(P)$, we have that $\widehat{T}(\hat{K}_c) \preccurlyeq \hat{K}_c$ and $\widehat{T}(\hat{K}_a) \preccurlyeq \hat{K}_a$. $\square$

11.6 Proposition Let P be the `while` program $\langle l_s \rangle$ `while` C `do` P_b `endwhile` $\langle l_f \rangle$, and let $\hat{K}$ be a family configuration such that $|\hat{K}| = P$. Denote by $\hat{K}_b$ the sub-configuration of $\hat{K}$ such that $|\hat{K}_b| = P_b$. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\widehat{T}(\hat{K}_b) \preccurlyeq \hat{K}_b$.

Proof: We have that $|\widehat{T}(\hat{K})| = |\hat{K}| = P$, which entails that $\mathit{labels}(\widehat{T}(\hat{K})) = \mathit{labels}(\hat{K}) = \mathit{labels}(P)$. Similarly, we have $\mathit{labels}(\widehat{T}(\hat{K}_b)) = \mathit{labels}(\hat{K}_b) = \mathit{labels}(P_b)$. Since P_b is a subprogram of P, if follows that $\mathit{labels}(P_b) \subseteq \mathit{labels}(P)$. Now, $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$ entails that $\widehat{T}(\hat{K})|_l \preccurlyeq \hat{K}|_l$, for all $l \in \mathit{labels}(P)$. Since $\mathit{labels}(P_b) \subseteq \mathit{labels}(P)$, we have that $\widehat{T}(\hat{K}_b) \preccurlyeq \hat{K}_b$. $\square$

11.7 Proposition Assume P is the skip program $\langle l_s \rangle$ `skip` $\langle l_f \rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration that covers K at l_s. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\hat{K}$ covers K at l_f.

Proof: The progression of P is $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0\rangle \,\texttt{skip}\, \langle l_f, \lambda\langle\rangle \,.\, \Sigma_0\rangle$. Let $\hat{K} = \langle l_s, \Phi_s\rangle \,\texttt{skip}\, \langle l_f, \Phi_f\rangle$. We have that $\widehat{T}(\hat{K}) = \langle l_s, \Phi_s\rangle \,\texttt{skip}\, \langle l_f, \Phi_s\rangle \preccurlyeq \hat{K}$, which entails that $\Phi_s \preccurlyeq \Phi_f$. Since Φ_s covers $\lambda\langle\rangle \,.\, \Sigma_0$, it follows that Φ_f covers $\lambda\langle\rangle \,.\, \Sigma_0$. □

11.8 Proposition Assume P is the skip program $\langle l_s\rangle\, x := E\, \langle l_f\rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration that covers K at l_s. If $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, then $\hat{K}$ covers K at l_f.

Proof: The progression of P is $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0\rangle\, x := E\, \langle l_f, \mathit{assign}(x, E, \lambda\langle\rangle \,.\, \Sigma_0\rangle$. Let $\hat{K} = \langle l_s, \Phi_s\rangle\, x := E\, \langle l_f, \Phi_f\rangle$. We have that $\widehat{T}(\hat{K}) = \langle l_s, \Phi_s\rangle\, x := E\, \langle l_f, \widehat{\mathit{assign}}(x, E, \Phi_s)\rangle \preccurlyeq \hat{K}$, which entails that $\widehat{\mathit{assign}}(x, E, \Phi_s) \preccurlyeq \Phi_f$. Since $\widehat{\mathit{assign}}(x, E, \Phi_s)$ covers $\widehat{\mathit{assign}}(x, E, \lambda\langle\rangle \,.\, \Sigma_0)$, it follows that Φ_f covers $\widehat{\mathit{assign}}(x, E, \lambda\langle\rangle \,.\, \Sigma_0)$. □

11.9 Proposition Assume P is the `if` program $\langle l_s\rangle \,\texttt{if}\; C \;\texttt{then}\; P_c \;\texttt{else}\; P_a \;\texttt{endif}\, \langle l_f\rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration such that $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$. The following statements hold.

a) If $\hat{K}$ covers K at l_s, then $\hat{K}$ covers K at $\mathit{first}(P_c)$ and $\mathit{first}(P_a)$.

b) If $\hat{K}$ covers K at $\mathit{last}(P_c)$ and $\mathit{last}(P_a)$, then $\hat{K}$ covers K at l_f.

Proof: We start by proving condition a). The progression of P is $K = \langle l_s, \lambda\langle\rangle \,.\, \Sigma_0\rangle$ $\texttt{if}\; C \;\texttt{then}\; K_c \;\texttt{else}\; K_a \;\texttt{endif}\; \langle l_f, \Psi_f\rangle$, where K_c is the progression of P_c w.r.t. $\{\sigma \,|\, \sigma \models C\} \cap \Sigma_0$, K_a is the progression of P_a w.r.t. $\{\sigma \,|\, \sigma \models \neg C\} \cap \Sigma_0$, and Ψ_f is an indexed set. It follows that $K_c|_{\mathit{first}(P_c)} = \mathit{filter}(C, \lambda\langle\rangle \,.\, \Sigma_0)$, and $K_a|_{\mathit{first}(P_a)} = \mathit{filter}(\neg C, \lambda\langle\rangle \,.\, \Sigma_0)$. Now, since $\hat{K}$ is a family configuration that covers K, we have that $\hat{K} = \langle l_s, \Phi_s\rangle \,\texttt{if}\; C \;\texttt{then}\; \hat{K}_c \;\texttt{else}\; \hat{K} \;\texttt{endif}\, \langle l_f, \Phi_f\rangle$,

where Φ_s covers $\lambda\langle\rangle\,.\,\Sigma_0$, $\hat{K}_c$ covers K_c, $\hat{K}_a$ covers K_a, and Φ_f covers Ψ_f. We also have that $\widehat{T}(\hat{K}) = \langle l_s, \Phi_s\rangle$ `if` C `then` $\widehat{\mathit{filter}}(C, \Phi_s)$ ⨟ $\widehat{T}(\hat{K}_c)$ `else` $\widehat{\mathit{filter}}(\neg C, \Phi_s)$ ⨟ $\widehat{T}(\hat{K})$ `endif` $\langle l_f, \hat{K}_c|_{\mathit{last}(P_c)} \mathbin{\widehat{\cup}} \hat{K}_a|_{\mathit{last}(P_a)}\rangle$. Obviously, $\widehat{\mathit{filter}}(C, \Phi_s)$ covers $\mathit{filter}(C, \lambda\langle\rangle\,.\,\Sigma_0)$, and $\widehat{\mathit{filter}}(\neg C, \Phi_s)$ covers $\mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\Sigma_0)$, which proves a).

Assume now that $\hat{K}_c|_{\mathit{last}(P_c)}$ covers $K_c|_{\mathit{last}(P_c)}$, and $\hat{K}_a|_{\mathit{last}(P_a)}$ covers $K_a|_{\mathit{last}(P_a)}$. Obviously, $\hat{K}_c|_{\mathit{last}(P_c)} \mathbin{\widehat{\cup}} \hat{K}_a|_{\mathit{last}(P_a)}$ covers $K_c|_{\mathit{last}(P_c)} \mathbin{\widehat{\cup}} K_a|_{\mathit{last}(P_a)}$, which proves b). □

11.12 Proposition Assume P is the `while` program $\langle l_s\rangle$ `while` C `do` P_b `endwhile` $\langle l_f\rangle$, and let Σ_0 be a set of start environments. Denote by K the progression of P w.r.t. Σ_0. Let $\hat{K}$ be a family configuration such that $\hat{K}$ covers K at l_s, and $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$. The following statements hold.

a) $\mathit{extr}(\hat{K}|_{\mathit{first}(P_b)}, 0)$ covers $\mathit{extr}(K|_{\mathit{first}(P_b)}, 0)$.

b) $\mathit{extr}(\hat{K}|_{\mathit{last}(P_b)}, \mu) \preccurlyeq \mathit{extr}(\hat{K}|_{\mathit{first}(P_b)}, \mu + 1)$.

c) If $\hat{K}|_{\mathit{last}(P_b)}$ covers $K|_{\mathit{last}(P_b)}$, then $\hat{K}|_{l_f}$ covers $K|_{l_f}$.

Proof: The progression of P w.r.t. Σ_0 is $K = \langle l_s, \lambda\langle\rangle\,.\,\Sigma_0\rangle$ `while` C `do` K_b `endwhile` $\langle l_f, \Psi_s\rangle$, where $\mathit{extr}(K_b, 0)$ is the progression of P_b w.r.t. $\Sigma_0 \cap \{\sigma \mid \sigma \models C\}$, $\mathit{extr}(K_b, \mu)$ is the progression of P_b w.r.t. $\mathit{extr}(K_b, \mu - 1) \cap \{\sigma \mid \sigma \models C\}$, for all $\mu > 0$, and Ψ_s is $\mathit{filter}(\neg C, \lambda\langle\rangle\,.\,\Sigma_0 \cup \mathit{collect}(K_b|_{\mathit{last}(P_b)}))$. Now, $\hat{K}$ must be of the form $\langle l_s, \Phi_s\rangle$ `while` C `do` $\hat{K}_b$ `endwhile` $\langle l_f, \Phi_f\rangle$, where Φ_s covers $\lambda\langle\rangle\,.\,\Sigma_0$. We have that $\widehat{T}(\hat{K}) = \langle l_s, \Phi_s\rangle$ `while` C `do` $\widehat{\mathit{filter}}(C, \widehat{\mathit{before}}(\Phi_s, \hat{K}_b|_{\mathit{last}(P_b)}))$ ⨟ $\widehat{T}(\hat{K}_b)$ `endwhile` $\langle l_f, \widehat{\mathit{filter}}(\neg C, \Phi_s \mathbin{\widehat{\cup}} \widehat{\mathit{collect}}(K_b|_{\mathit{last}(P_b)}))\rangle$. Using Remarks 11.10 and 11.11 it follows immediately that $\mathit{extr}(\widehat{T}(\hat{K})|_{\mathit{first}(P_b)}, 0)$ covers $\mathit{extr}(K|_{\mathit{first}(P_b)}, 0)$,

and using the fact that $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$, we prove a). In order to prove b), we note that $extr(\hat{K}|_{last(P_b)}, \mu) = extr(\widehat{T}(\hat{K})|_{first(P_b)}, \mu{+}1)$, for all $\mu \in \mathbb{N}$. Then, from $\widehat{T}(\hat{K}) \preccurlyeq \hat{K}$ we infer b). Statement c) follows immediately from the fact that if $\hat{K}|_{last(P_b)}$ covers $K|_{last(P_b)}$, then $\widehat{filter}(C, \widehat{collect}(\hat{K}|_{last(P_b)}))$ covers $filter(C, collect(K|_{last(P_b)}))$, for all program constraints C. $\square$

13.3 Proposition For all program variables x, program expressions E, and formulas $\mathcal{F} \in \mathrm{LAL}$, we have $\widehat{assign}(x, E, [\![\mathcal{F}]\!]) \subseteq [\![Assign(x, E, \mathcal{F})]\!]$.

Proof: We show that the proposition holds for all the cases in which the $Assign$ meta-operator was defined.

Case 1: $\mathcal{F} = \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi$, for some $k \geq 0$. In this case we have

$$\widehat{assign}(x, E, [\![\mathcal{F}]\!]) = \lceil \{ assign(x, E, \Psi) \mid \Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi]\!] \rfloor \} \rceil$$

$$\begin{aligned} [\![Assign(x, E, \mathcal{F})]\!] &= [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \exists \delta \,.\, (\varphi[\delta/x] \wedge x = (E[\delta/x]))]\!] \\ &= \lambda\langle \mu_1 \cdots \mu_k \rangle . \{ \Sigma \mid \text{for all } \sigma \in \Sigma, \\ &\qquad \sigma \models \exists \delta . \varphi[\delta/x, \mu_1/\nu_1, \ldots, \mu_1/\nu_1] \wedge x = (E[\delta/x]) \} \end{aligned}$$

By using Remarks 13.1 and 13.2, it follows that, for a given $\tilde{\mu}$,

$$\begin{aligned} \widehat{assign}(x, E, [\![\mathcal{F}]\!])(\tilde{\mu}) &= \lceil \{ assign(x, E, \Psi(\tilde{\mu})) \mid \Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi]\!] \rfloor \} \rceil \\ &= \{ \{ \sigma \mid \text{exists } \sigma' \in \Psi(\tilde{\mu}) \text{ s.t. } \sigma = \sigma[x \mapsto E(\sigma)] \} \mid \\ &\qquad \Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi]\!] \rfloor \} \end{aligned}$$

$$[\![Assign(x, E, \mathcal{F})]\!] = \begin{cases} \emptyset, & \text{if } size(\tilde{\mu}) \neq k \\ \{ \Sigma \mid \text{for all } \sigma \in \Sigma, & \\ \qquad \sigma \models \exists \delta . \varphi[\delta/x, \mu_1/\nu_1, \ldots, \mu_1/\nu_1] \wedge x = (E[\delta/x]) \} & \\ \qquad\qquad \text{otherwise} & \end{cases}$$

We now have two subcases: when $size(\tilde{\mu}) \neq k$ and when $size(\tilde{\mu}) = k$. Assume first that $size(\tilde{\mu}) \neq k$. The constraint $\Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi]\!] \rfloor$ implies $\Psi(\tilde{\mu}) = \emptyset$, and as a result, $\widehat{assign}(x, E, [\![\mathcal{F}]\!])(\tilde{\mu}) \subseteq [\![Assign(x, E, \mathcal{F})]\!](\tilde{\mu})$. Assume now that $\tilde{\mu} = \mu_1 \cdots \mu_k$. In this case, the constraint $\Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi]\!] \rfloor$ implies that $\Psi(\mu_1 \cdots \mu_k) = \{\sigma \,|\, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$. It follows that $\widehat{assign}(x, E, [\![\mathcal{F}]\!])(\tilde{\mu}) = \{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models \exists \delta . \varphi[\delta/x, \mu_1/\nu_1, \ldots, \mu_1/\nu_1] \wedge x = (E[\delta/x])\} \subseteq [\![Assign(x, E, \mathcal{F})]\!](\tilde{\mu})$. This proves the first case.

Case 2: $\mathcal{F} = \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi^*$, for some $k \geq 0$. In this case we have

$$\widehat{assign}(x, E, [\![\mathcal{F}]\!]) \;=\; \lceil \{ assign(x, E, \Psi) \,|\, \Psi \in \lfloor [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi^*]\!] \rfloor \} \rceil$$

$$[\![Assign(x, E, \mathcal{F})]\!] \;=\; [\![\lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, (\exists \delta \,.\, (\varphi[\delta/x] \wedge x = (E[\delta/x])))^*]\!]$$

$$= \lambda\langle \mu_1 \cdots \mu_k \rangle . \begin{cases} \emptyset, & \text{if } \models \forall(\neg \exists \delta . \varphi[\delta/x, \mu_1/\nu_1, \ldots, \mu_k/\nu_k] \wedge \\ & \qquad\qquad x = (E[\delta/x])) \\ \{\Sigma \,|\, \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, & \\ \qquad \sigma \models \exists \delta . \varphi[\delta/x, \mu_1/\nu_1, \ldots, \mu_k/\nu_k] \wedge x = (E[\delta/x])\} & \\ \qquad\qquad \text{otherwise} \end{cases}$$

By using Remarks 13.1 and 13.2, it follows that, for a given $\tilde{\mu}$,

$$\begin{aligned}\widehat{assign}(x,E,[\![\mathcal{F}]\!])(\tilde{\mu}) &= \lceil\{assign(x,E,\Psi(\tilde{\mu}))\,|\,\Psi\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi^*]\!]\rfloor\}\rceil\\ &= \{\{\sigma\,|\,\text{exists } \sigma'\in\Psi(\tilde{\mu}) \text{ s.t. } \sigma=\sigma[x\mapsto E(\sigma)]\}\,|\\ &\qquad\qquad \Psi\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi^*]\!]\rfloor\}\end{aligned}$$

$$[\![Assign(x,E,\mathcal{F})]\!] = \begin{cases}\emptyset, & \text{if } size(\tilde{\mu})\neq k \text{ or}\\ & \qquad \models\forall(\neg\exists\delta.\varphi[\delta/x,\mu_1/\nu_1,\ldots,\mu_k/\nu_k]\wedge\\ & \qquad\qquad x=(E[\delta/x]))\\ \{\Sigma\,|\,\text{for all } \sigma\in\Sigma, & \\ \qquad \sigma\models\exists\delta.\varphi[\delta/x,\mu_1/\nu_1,\ldots,\mu_k/\nu_k]\wedge x=(E[\delta/x])\} & \\ \qquad\qquad \text{otherwise}\end{cases}$$

We now have two subcases: when $size(\tilde{\mu})\neq k$ and when $size(\tilde{\mu})=k$. Assume first that $size(\tilde{\mu})\neq k$. The constraint $\Psi\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi]\!]\rfloor$ implies $\Psi(\tilde{\mu})=\emptyset$, and as a result, $\widehat{assign}(x,E,[\![\mathcal{F}]\!])(\tilde{\mu})\subseteq[\![Assign(x,E,\mathcal{F})]\!](\tilde{\mu})$. Assume now that $\tilde{\mu}=\mu_1\cdots\mu_k$. In this case, the constraint $\Psi\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi^*]\!]\rfloor$ implies that $\Psi(\mu_1\cdots\mu_k)=\{\sigma\,|\,\sigma\models\varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k]\}$ and $\Psi\neq\emptyset$. It follows that $\widehat{assign}(x,E,[\![\mathcal{F}]\!])(\tilde{\mu})=\{\Sigma\,|\,\text{for all } \sigma\in\Sigma,\sigma\models\exists\delta.\varphi[\delta/x,\mu_1/\nu_1,\ldots,\mu_1/\nu_1]\wedge x=(E[\delta/x])\}\subseteq[\![Assign(x,E,\mathcal{F})]\!](\tilde{\mu})$. This proves the proposition. □

13.4 Proposition For all program constraints C, and formulas $\mathcal{F}\in$ LAL, we have $\widehat{filter}(C,[\![\mathcal{F}]\!])\subseteq[\![Filter(C,\mathcal{F})]\!]$.

Proof: We show that the proposition holds for all the cases in which the *Filter* meta-operator was defined.

Case 1: $\mathcal{F} = \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi$, for some $k \geq 0$. In this case we have

$$\begin{aligned}
\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) &= \lceil \{\mathit{filter}(C, \Psi) \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] \rfloor\} \rceil \\
[\![\mathit{Filter}(C, \mathcal{F})]\!] &= \lambda\langle\mu_1 \cdots \mu_k\rangle \,.\, \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}
\end{aligned}$$

By applying Remark 13.2, it follows that, for a given μ,

$$\begin{aligned}
\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!])(\tilde{\mu}) &= \{(\mathit{filter}(C, \Psi)(\tilde{\mu}) \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] \rfloor\} \\
&= \{\{\sigma \mid \sigma \in \Psi(\tilde{\mu}) \text{ and } \sigma \models C\} \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] \rfloor\}
\end{aligned}$$

$$[\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu}) = \begin{cases} \{\emptyset\}, & \text{if } \tilde{\mu} \neq k \\ \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k] \wedge C, \\ \qquad\qquad \text{for all } \tilde{\mu} = \mu_1 \cdots \mu_k \end{cases}$$

We now have two subcases: when $\mathit{size}(\tilde{\mu}) \neq k$ and when $\mathit{size}(\tilde{\mu}) = k$. Assume first that $\mathit{size}(\tilde{\mu}) \neq k$. The constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] \rfloor$ implies $\Psi(\tilde{\mu}) = \emptyset$, and as a result, $\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!])(\tilde{\mu}) = \emptyset$. It follows that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$. Assume now that $\mathit{size}(\tilde{\mu}) = k$, and that $\tilde{\mu} = \mu_1 \cdots \mu_k$. In this case the constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi]\!] \rfloor$ entails that

$$\Psi(\tilde{\mu}) \subseteq \{\sigma \mid \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$$

From this, it follows that

$$\begin{aligned}
\mathit{filter}(C, [\![\mathcal{F}]\!])(\mu_1 \cdots \mu_k) &= \{\{\sigma \mid \sigma \in \Sigma \text{ and } \sigma \models C\} \mid \Sigma \subseteq \{\sigma \mid \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}\} \\
&= \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models C \text{ and } \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}
\end{aligned}$$

From this we infer that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$. The two subcases have proved that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$ for all $\tilde{\mu} \in \mathbf{Idx}$, which entails that $\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!]$.

Case 2: $\mathcal{F} = \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*$ and $\models (\forall(C \to \varphi)) \vee (\forall(C \to \neg\varphi))$, for some $k \geq 0$. In this case we have

$$\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) \;=\; \lceil \{\mathit{filter}(C, \Psi) \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*]\!] \rfloor\} \rceil$$

$$[\![\mathit{Filter}(C, \mathcal{F})]\!] \;=\; \lambda\langle\mu_1 \cdots \mu_k\rangle \,. \begin{cases} \{\emptyset\}, & \text{if } \models \forall(\neg(C \wedge \varphi[\mu_1/\nu_1, \ldots \mu_k/\nu_k])) \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and forall } \sigma \in \Sigma, & \\ \quad \sigma \models (C \wedge \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k])\}, & \\ & \text{otherwise} \end{cases}$$

By applying Remark 13.2, it follows that, for a given μ,

$$\begin{aligned} \widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!])(\tilde{\mu}) &= \{(\mathit{filter}(C, \Psi)(\tilde{\mu}) \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*]\!] \rfloor\} \\ &= \{\{\sigma \mid \sigma \in \Psi(\tilde{\mu}) \text{ and } \sigma \models C\} \mid \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi^*]\!] \rfloor\} \end{aligned}$$

$$[\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu}) \;=\; \begin{cases} \{\emptyset\}, & \text{if } \models \forall(\neg(C \wedge \varphi[\mu_1/\nu_1, \ldots \mu_k/\nu_k])) \text{ and } \mathit{size}(\tilde{\mu}) \neq k \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and forall } \sigma \in \Sigma, & \\ \quad \sigma \models (C \wedge \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k])\}, & \\ & \text{otherwise} \end{cases}$$

Again, we have two subcases. The first one is when $\mathit{size}(\tilde{\mu}) \neq k$ or $\forall(C \to \neg\varphi)$. In this case, it is easy to see that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) = [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu}) = \emptyset$. The second sub-case is when $\mathit{size}(\tilde{\mu}) \neq k$ and $\forall(C \to \varphi)$. In this case we have

$$(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) = [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu}) = \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}.$$

From the two subcases it follows that $\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!]$.

Case 3: $\mathcal{F} = \lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi^*$ and $\models (\neg\forall(C \to \varphi)) \vee (\forall(C \to \neg\varphi))$, for some $k \geq 0$.

In this case we have

$$\begin{aligned}
\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) &= \lceil \{\mathit{filter}(C, \Psi) \,|\, \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi^*]\!] \rfloor\} \rceil \\
[\![\mathit{Filter}(C, \mathcal{F})]\!] &= \lambda\langle\mu_1 \cdots \mu_k\rangle\,.\,\{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}
\end{aligned}$$

By applying Remark 13.2, it follows that, for a given μ,

$$\begin{aligned}
\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!])(\tilde{\mu}) &= \{(\mathit{filter}(C, \Psi)(\tilde{\mu}) \,|\, \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi^*]\!] \rfloor\} \\
&= \{\{\sigma \,|\, \sigma \in \Psi(\tilde{\mu}) \text{ and } \sigma \models C\} \,|\, \Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi^*]\!] \rfloor\}
\end{aligned}$$

$$[\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu}) = \begin{cases} \{\emptyset\}, & \text{if } \tilde{\mu} \neq k \\ \{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k] \wedge C, \\ \qquad\qquad \text{for all } \tilde{\mu} = \mu_1 \cdots \mu_k \end{cases}$$

The constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi^*]\!] \rfloor$ may imply that Ψ is non-empty for certain values of $\tilde{\mu}$. However, since $\models \neg\forall(C \to \varphi)$, $\mathit{filter}(C, \Psi)(\tilde{\mu})$ may not necessarily be non-empty. We now have two subcases: when $\mathit{size}(\tilde{\mu}) \neq k$ and when $\mathit{size}(\tilde{\mu}) = k$. Assume first that $\mathit{size}(\tilde{\mu}) \neq k$. The constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi]\!] \rfloor$ implies $\Psi(\tilde{\mu}) = \emptyset$, and as a result, $\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!])(\tilde{\mu}) = \emptyset$. It follows that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$. Assume now that $\mathit{size}(\tilde{\mu}) = k$, and that $\tilde{\mu} = \mu_1 \cdots \mu_k$. In this case the constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle\,.\,\varphi]\!] \rfloor$ entails that

$$\Psi(\tilde{\mu}) \subseteq \{\sigma \,|\, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$$

From this, it follows that

$$\begin{aligned}
\mathit{filter}(C, [\![\mathcal{F}]\!])(\mu_1 \cdots \mu_k) &= \{\{\sigma \,|\, \sigma \in \Sigma \text{ and } \sigma \models C\} \,|\, \Sigma \subseteq \{\sigma \,|\, \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}\} \\
&= \{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models C \text{ and } \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}
\end{aligned}$$

We infer that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$. The two subcases have proved that $(\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!](\tilde{\mu})$ for all $\tilde{\mu} \in \mathbf{Idx}$, which entails that $\widehat{\mathit{filter}}(C, [\![\mathcal{F}]\!]) \subseteq [\![\mathit{Filter}(C, \mathcal{F})]\!]$. $\square$

13.5 Proposition For all formulas $\mathcal{F}_1, \mathcal{F}_2 \in \text{LAL}$, we have $\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]) \subseteq [\![Before(\mathcal{F}_1, \mathcal{F}_2)]\!]$.

Proof: We prove that the proposition holds for each of the cases of the definition of *Before*.

Case 1: $\mathcal{F}_1 = \lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1$ and $\mathcal{F}_2 = \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2$, for some $k \geq 0$. In this case we have:

$$\begin{aligned}
\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]) &= \lceil \{ before(\Psi_1, \Psi_2) \mid \Psi_1 \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1]\!] \rfloor \text{ and} \\
&\qquad \Psi_2 \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi_2]\!] \rfloor \} \rceil \\
\\
[\![Before(\mathcal{F}_1, \mathcal{F}_2)]\!] &= [\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle.\{((\nu_{k+1} = 0 \to \varphi_1) \wedge \\
&\qquad ((\nu_{k+1} > 0 \to \varphi_2[(\nu_{k+1} - 1)/\nu_{k+1}]))\}]\!] \\
&= \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle. \\
&\qquad \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \forall((\mu_{k+1} = 0 \to \varphi_1[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]) \wedge \\
&\qquad\quad (\mu_{k+1} > 0 \to \varphi_2[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, (\mu_{k+1} - 1)/\nu_{k+1}]))\}
\end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the *before* operator given in Figure 7.3. it follows that, for a given μ,

$$\begin{aligned}
(\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]))(\epsilon) &= \{\emptyset\} \\
(\widehat{before}([\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!]))(\tilde{\mu}0) &= \{\Psi_1(\tilde{\mu}) \mid \Psi_1 \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \varphi_1]\!] \rfloor\} \\
(\widehat{before}(\mathcal{F}_1, \mathcal{F}_2))(\tilde{\mu}\mu') &= \{\Psi_2(\tilde{\mu}(\mu' - 1)) \mid \Psi_2 \in \lfloor [\![\lambda\langle\nu_1 \cdots \nu_{k+1}\rangle \,.\, \varphi_2]\!] \rfloor\}
\end{aligned}$$

$$[\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\epsilon) = \{\emptyset\}$$

$$[\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}\mu') = \begin{cases} \{\emptyset\}, & \text{if } \mathit{size}(\tilde{\mu}) \neq k \\ \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \forall(\varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k])\}, & \\ \qquad \text{if } \tilde{\mu} = \mu_1\cdots\mu_k \text{ and } \mu' = 0 & \\ \{\Sigma \mid \text{for all } \sigma \in \Sigma, & \\ \qquad \sigma \models \forall(\varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,(\mu_{k+1}-1)/\nu_{k+1}])\}, & \\ \qquad \text{if } \tilde{\mu} = \mu_1\cdots\mu_k \text{ and } \mu' > 0 & \end{cases}$$

We use the relations above to verify that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu})$, for all $\tilde{\mu} \in \mathbf{Idx}$. The statement clearly holds for $\mathit{size}(\tilde{\mu}) \neq k+1$. In order to verify that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}0) \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}0)$. we note that the constraint $\Psi_1 \in \lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1]\!]\rfloor$ entails that $\Psi_1(\mu_1\cdots\mu_k) \subseteq \{\sigma \mid \sigma \models \forall(\varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k])\}$. We also note that the constraint $\Psi_2 \in \lfloor[\![\lambda\langle\nu_1\cdots\nu_{k+1}\rangle\,.\,\varphi_2]\!]\rfloor$ entails that $\Psi_2(\mu_1\cdots\mu_k\mu_{k+1}) \subseteq \{\sigma \mid \sigma \models \forall(\varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\mu_{k+1}/\nu_{k+1}])\}$. This proves that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}\mu') \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}\mu')$, where $\mathit{size}(\tilde{\mu}) = k$ and $\mu' > 0$.

Case 2: $\mathcal{F}_1 = \lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1^*$ and $\mathcal{F}_2 = \lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2^*$, for some $k \geq 0$. For convenience we denote by $B(\mu_1\cdots\mu_k)$ the first-order formula $\forall((\mu_{k+1}=0 \rightarrow \varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k]) \wedge (\mu_{k+1}>0 \rightarrow \varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,(\mu_{k+1}-1)/\nu_{k+1}]))$. In

this case we have:

$$\begin{aligned}\widehat{before}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]) &= \lceil\{before(\Psi_1,\Psi_2)\mid \Psi_1\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1^*]\!]\rfloor \text{ and}\\ &\qquad\qquad \Psi_2\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2^*]\!]\rfloor\}\rceil\end{aligned}$$

$$\begin{aligned}[\![Before(\mathcal{F}_1,\mathcal{F}_2)]\!] &= [\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle.\{((\nu_{k+1}=0\rightarrow\varphi_1)\wedge\\ &\qquad\qquad ((\nu_{k+1}>0\rightarrow\varphi_2[(\nu_{k+1}-1)/\nu_{k+1}]))^*\}]\!]\\ &= \lambda\langle\mu_1\cdots\mu_k\mu_{k+1}\rangle.\begin{cases}\{\emptyset\}, & \text{if } \models\neg B(\mu_1\cdots\mu_k)\\ \{\Sigma\mid \Sigma\neq\emptyset \text{ and for all } \sigma\in\Sigma,\\ \quad \sigma\models\forall((\mu_{k+1}=0\rightarrow\\ \quad\varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k])\wedge\\ \quad(\mu_{k+1}>0\rightarrow\\ \quad\varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,(\mu_{k+1}-1)/\nu_{k+1}]))\}, & \\ \quad\text{otherwise} & \end{cases}\end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the *before* operator given in Figure 7.3. it follows that, for a given μ,

$$\begin{aligned}(\widehat{before}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\epsilon) &= \{\emptyset\}\\ (\widehat{before}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}0) &= \{\Psi_1(\tilde{\mu})\mid\Psi_1\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1^*]\!]\rfloor\}\\ (\widehat{before}(\mathcal{F}_1,\mathcal{F}_2))(\tilde{\mu}\mu') &= \{\Psi_2(\tilde{\mu}(\mu'-1))\mid\Psi_2\in\lfloor[\![\lambda\langle\nu_1\cdots\nu_{k+1}\rangle\,.\,\varphi_2^*]\!]\rfloor\}\end{aligned}$$

$$[\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\epsilon) = \{\emptyset\}$$

$$[\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}\mu') = \begin{cases} \{\emptyset\}, & \text{if } \mathit{size}(\tilde{\mu}) \neq k \text{ and } \models \neg B(\tilde{\mu}) \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, \sigma \models \forall(\varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k])\}, & \\ \qquad \text{if } \tilde{\mu} = \mu_1\cdots\mu_k \text{ and } \mu' = 0 & \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, & \\ \qquad \sigma \models \forall(\varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,(\mu_{k+1}-1)/\nu_{k+1}])\}, & \\ \qquad \text{if } \tilde{\mu} = \mu_1\cdots\mu_k \text{ and } \mu' > 0 & \end{cases}$$

We use the relations above to verify that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}) \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu})$, for all $\tilde{\mu} \in \mathbf{Idx}$. The statement clearly holds for $\mathit{size}(\tilde{\mu}) \neq k+1$. In order to verify that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}0) \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}0)$. we note that the constraint $\Psi_1 \in \lfloor[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1^*]\!]\rfloor$ entails that $\Psi_1(\mu_1\cdots\mu_k) \subseteq \{\sigma \mid \sigma \models \forall(\varphi_1[\mu_1/\nu_1,\ldots,\mu_k/\nu_k])\}$. We also note that the constraint $\Psi_2 \in \lfloor[\![\lambda\langle\nu_1\cdots\nu_{k+1}\rangle\,.\,\varphi_2^*]\!]\rfloor$ entails that $\Psi_2(\mu_1\cdots\mu_k\mu_{k+1}) \subseteq \{\sigma \mid \sigma \models \forall(\varphi_2[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\mu_{k+1}/\nu_{k+1}])\}$. This proves that $(\widehat{\mathit{before}}([\![\mathcal{F}_1]\!],[\![\mathcal{F}_2]\!]))(\tilde{\mu}\mu') \subseteq [\![\mathit{Before}(\mathcal{F}_1,\mathcal{F}_2)]\!](\tilde{\mu}\mu')$, where $\mathit{size}(\tilde{\mu}) = k$ and $\mu' > 0$.In order to prove the last three cases, we note that $[\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi^*]\!] \subseteq [\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi]\!]$, for all $k \geq 0$ and simple formula φ. It follows that

$$\widehat{\mathit{before}}([\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1^*]\!],[\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2]\!]) \subseteq \widehat{\mathit{before}}([\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1]\!],[\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2]\!])$$

and

$$\widehat{\mathit{before}}([\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1]\!],[\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2^*]\!]) \subseteq \widehat{\mathit{before}}([\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1]\!],[\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi_2]\!]).$$

The result follows immediately. □

13.6 Proposition For all formulas $\mathcal{F} \in$ LAL, we have $\widehat{\mathit{collect}}([\![\mathcal{F}]\!]) \subseteq [\![\mathit{Collect}(\mathcal{F})]\!]$.

Proof: We prove that the proposition holds for each of the cases of the definition of *Collect*.

Case 1: $\mathcal{F} = \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi$. We have that

$$\begin{aligned}
\widehat{collect}([\![\mathcal{F}]\!]) &= \lceil\{collect(\Psi) \mid \Psi \in \lfloor[\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi]\!]\rfloor\}\rceil \\
[\![Collect(\mathcal{F})]\!] &= [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \exists\delta.\varphi[\delta/\nu_{k+1}]]\!] \\
&= \lambda\langle\mu_1 \cdots \mu_k\rangle \,..\{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \exists\delta.\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \delta/\nu_{k+1}]\}
\end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the *collect* operator given in Figure 7.3. it follows that, for a given μ,

$$(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) = \{\textstyle\bigcup_{\mu' \geq 0} \Psi(\tilde{\mu}\mu') \mid \Psi \in \lfloor[\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi]\!]\rfloor\}$$

$$[\![\widehat{collect}(\mathcal{F})]\!](\tilde{\mu}) = \begin{cases} \{\emptyset\}, & \text{if } size(\tilde{\mu}) \neq k \\ \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \exists\delta.(\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \delta/\nu_{k+1}])\} & \\ \qquad\qquad \text{otherwise} & \end{cases}$$

We now have two subcases: either $size(\tilde{\mu}) \neq k$, or $size(\tilde{\mu}) = k$. In case $size(\tilde{\mu}) \neq k$, we have that $(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) = [\![Collect(\mathcal{F})]\!](\tilde{\mu}) = \{\emptyset\}$. In case $size(\tilde{\mu}) = k$, from the constraint $\Psi \in \lfloor[\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi]\!]\rfloor$ it follows that $\Psi(\tilde{\mu}\mu') \subseteq \{\sigma \mid \sigma \models \varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \mu_{k+1}/\nu_{k+1}]\}$. Using this, we get that $(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq \{\Sigma \mid \text{for all } \sigma \in \Sigma, \sigma \models \exists\delta.(\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \delta/\nu_{k+1}])\} = [\![Collect(\mathcal{F})]\!](\tilde{\mu})$.

Case 2: $\mathcal{F} = \lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi^*$. We have that

$$\begin{aligned}
\widehat{collect}([\![\mathcal{F}]\!]) &= \lceil\{collect(\Psi) \mid \Psi \in \lfloor[\![\lambda\langle\nu_1 \cdots \nu_k\nu_{k+1}\rangle \,.\, \varphi^*]\!]\rfloor\}\rceil \\
[\![Collect(\mathcal{F})]\!] &= [\![\lambda\langle\nu_1 \cdots \nu_k\rangle \,.\, \exists\delta.\varphi[\delta/\nu_{k+1}]]\!] \\
&= \lambda\langle\mu_1 \cdots \mu_k\rangle.\begin{cases} \{\emptyset\}, & \text{if } \models \forall(\neg\exists\delta.\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \delta/\nu_{k+1}]) \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, & \\ \qquad \sigma \models \exists\delta.\varphi[\mu_1/\nu_1, \ldots, \mu_k/\nu_k, \delta/\nu_{k+1}]\} & \\ \qquad\qquad \text{otherwise} & \end{cases}
\end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the *collect* operator given in Figure 7.3. it follows that, for a given μ,

$$(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) \;=\; \{\textstyle\bigcup_{\mu'\geq 0}\Psi(\tilde{\mu}\mu') \mid \Psi \in \lfloor [\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi^*]\!]\rfloor\}$$

$$[\![\widehat{collect}(\mathcal{F})]\!](\tilde{\mu}) \;=\; \begin{cases} \{\emptyset\}, & \text{if } size(\tilde{\mu}) \neq k \text{ or} \\ & \quad \models \forall(\neg\exists\delta.\varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\delta/\nu_{k+1}]) \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma\in\Sigma, & \\ \qquad \sigma \models \exists\delta.(\varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\delta/\nu_{k+1}])\} & \\ \qquad \text{otherwise} \end{cases}$$

We now have two subcases. The first is when $size(\tilde{\mu}) \neq k$, or $\models \forall(\neg\exists\delta.\varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\delta/\nu_{k+1}])$. In this case, we have that $(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) = [\![Collect(\mathcal{F})]\!](\tilde{\mu}) = \{\emptyset\}$. In case $size(\tilde{\mu}) = k$, from the constraint $\Psi \in \lfloor [\![\lambda\langle\nu_1\cdots\nu_k\nu_{k+1}\rangle\,.\,\varphi]\!]\rfloor$ it follows that $\Psi(\tilde{\mu}\mu') \subseteq \{\sigma \mid \sigma \models \varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\mu_{k+1}/\nu_{k+1}]\}$ and $\Psi \neq \emptyset$. Using this, we get that $(\widehat{collect}([\![\mathcal{F}]\!]))(\tilde{\mu}) \subseteq \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, \sigma \models \exists\delta.(\varphi[\mu_1/\nu_1,\ldots,\mu_k/\nu_k,\delta/\nu_{k+1}])\} = [\![Collect(\mathcal{F})]\!](\tilde{\mu})$. $\square$

13.7 Proposition For all formulas $\mathcal{F}_1, \mathcal{F}_2 \in$ LAL, we have $[\![\mathcal{F}_1]\!]\,\widehat{\cup}\,[\![\mathcal{F}_2]\!] \subseteq [\![F_1 \sqcup F_2]\!]$.

Proof: We prove that the proposition holds for each of the cases of the definition of $\sqcup$.

Case 1: $\mathcal{F}_1 = \lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1$ and $\mathcal{F}_2 = \lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_2$. We have that

$$[\![\mathcal{F}_1]\!]\,\widehat{\cup}\,[\![\mathcal{F}_2]\!] \;=\; \lceil\{\Psi_1\cup\Psi_2 \mid \Psi_1 \in \lfloor [\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1]\!]\rfloor \text{ and } \Psi_2 \in \lfloor [\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_2]\!]\rfloor\}\rceil$$

$$\begin{aligned} [\![\mathcal{F}_1 \sqcup \mathcal{F}_2]\!] &= [\![\lambda\langle\nu_1\cdots\nu_k\rangle\,.\,\varphi_1\vee\varphi_2]\!] \\ &= \lambda\langle\mu_1\cdots\mu_k\rangle\,.\,\{\Sigma \mid \text{for all } \sigma\in\Sigma, \sigma \models (\varphi_1\vee\varphi_2)[\mu_1/\nu_1,\ldots,\mu_k/\nu_k]\} \end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the $\cup$ operator given in Figure 7.3. it follows that, for a given μ,

$$(\llbracket \mathcal{F}_1 \sqcup \mathcal{F}_2 \rrbracket)(\tilde{\mu}) = \{\Psi_1(\tilde{\mu}) \cup \Psi_2(\tilde{\mu}) \,|\, \Psi_1 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1 \rrbracket \rfloor \text{ and } \Psi_2 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2 \rrbracket \rfloor\}$$

$$\llbracket \mathcal{F}_1 \sqcup \mathcal{F}_2 \rrbracket(\tilde{\mu}) = \begin{cases} \{\emptyset\}, & \text{if } \mathit{size}(\tilde{\mu}) \neq k \\ \{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models (\varphi_1 \vee \varphi_2)[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}, & \\ & \text{otherwise} \end{cases}$$

We now have two cases. If $\mathit{size}(\tilde{\mu}) \neq k$, then $\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket = \llbracket F_1 \sqcup F_2 \rrbracket = \{\emptyset\}$. In case $\mathit{size}(\tilde{\mu}) = k$, from the constraints $\Psi_1 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1} \rangle \,.\, \varphi_1 \rrbracket \rfloor$ and $\Psi_2 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1} \rangle \,.\, \varphi_2 \rrbracket \rfloor$ it follows that $\Psi_1(\tilde{\mu}) \subseteq \{\sigma \,|\, \sigma \models \varphi_1[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$ and $\Psi_2(\tilde{\mu}) \subseteq \{\sigma \,|\, \sigma \models \varphi_2[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$. Using this, we get that $(\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket)(\tilde{\mu}) \subseteq \{\Sigma \,|\, \text{for all } \sigma \in \Sigma, \sigma \models (\varphi_1 \vee \varphi_2)[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\} = \llbracket F_1 \sqcup F_2 \rrbracket(\tilde{\mu})$. From the two subcases, it follows that $(\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket)(\tilde{\mu}) \subseteq \llbracket F_1 \sqcup F_2 \rrbracket(\tilde{\mu})$, for all $\tilde{\mu} \in \mathbf{Idx}$.

Case 2: $\mathcal{F}_1 = \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1^*$ and $\mathcal{F}_2 = \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2^*$. We have that

$$\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket = \lceil \{\Psi_1 \cup \Psi_2 \,|\, \Psi_1 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1^* \rrbracket \rfloor \text{ and } \Psi_2 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2^* \rrbracket \rfloor\} \rceil$$

$$\begin{aligned} \llbracket \mathcal{F}_1 \sqcup \mathcal{F}_2 \rrbracket &= \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, (\varphi_1 \vee \varphi_2)^* \rrbracket \\ &= \lambda\langle \mu_1 \cdots \mu_k \rangle . \begin{cases} \{\emptyset\}, & \text{if } \models \forall(\neg(\varphi_1 \vee \varphi_2)) \\ \{\Sigma \,|\, \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, & \\ \qquad \sigma \models (\varphi_1 \vee \varphi_2)[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}, & \\ & \text{otherwise} \end{cases} \end{aligned}$$

By applying Remarks 13.2 and 13.2, and the definition of the $\cup$ operator given in Figure 7.3. it follows that, for a given μ,

$$\begin{aligned}(\llbracket \mathcal{F}_1 \sqcup \mathcal{F}_2 \rrbracket)(\tilde{\mu}) &= \{\Psi_1(\tilde{\mu}) \cup \Psi_2(\tilde{\mu}) \mid \Psi_1 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1^* \rrbracket \rfloor \text{ and} \\ &\qquad \Psi_2 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2^* \rrbracket \rfloor \}\end{aligned}$$

$$\llbracket \mathcal{F}_1 \sqcup \mathcal{F}_2 \rrbracket(\tilde{\mu}) = \begin{cases} \{\emptyset\}, & \text{if } \mathit{size}(\tilde{\mu}) \neq k \text{ or } \models \forall(\neg(\varphi_1 \vee \varphi_2)) \\ \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, & \\ \qquad \sigma \models (\varphi_1 \vee \varphi_2)[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}, & \\ \text{otherwise} & \end{cases}$$

We now have two cases. If $\mathit{size}(\tilde{\mu}) \neq k$, or $\models \forall(\neg(\varphi_1 \vee \varphi_2))$, then $\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket = \llbracket F_1 \sqcup F_2 \rrbracket = \{\emptyset\}$. In case $\mathit{size}(\tilde{\mu}) = k$, and $\models \exists(\varphi_1 \vee \varphi_2)$, from the constraints $\Psi_1 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1} \rangle \,.\, \varphi_1^* \rrbracket \rfloor$ and $\Psi_2 \in \lfloor \llbracket \lambda\langle \nu_1 \cdots \nu_k \nu_{k+1} \rangle \,.\, \varphi_2^* \rrbracket \rfloor$ it follows that $\Psi_1(\tilde{\mu}) \subseteq \{\sigma \mid \sigma \models \varphi_1[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$ and $\Psi_2(\tilde{\mu}) \subseteq \{\sigma \mid \sigma \models \varphi_2[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\}$, and also $\Psi_1 \neq \emptyset$ and $\Psi_2 \neq \emptyset$. From this, we get that $(\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket)(\tilde{\mu}) \subseteq \{\Sigma \mid \Sigma \neq \emptyset \text{ and for all } \sigma \in \Sigma, \sigma \models (\varphi_1 \vee \varphi_2)[\mu_1/\nu_1, \ldots, \mu_k/\nu_k]\} = \llbracket F_1 \sqcup F_2 \rrbracket(\tilde{\mu})$. From the two subcases, it follows that $(\llbracket \mathcal{F}_1 \rrbracket \,\widehat{\cup}\, \llbracket \mathcal{F}_2 \rrbracket)(\tilde{\mu}) \subseteq \llbracket F_1 \sqcup F_2 \rrbracket(\tilde{\mu})$, for all $\tilde{\mu} \in \mathbf{Idx}$.

In order to prove the last three cases, we note that $\llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi^* \rrbracket \subseteq \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi \rrbracket$, for all $k \geq 0$ and simple formula φ. It follows that

$$\begin{aligned}&\llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1^* \rrbracket \sqcup \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2 \rrbracket) \subseteq \\ &\qquad \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1 \rrbracket \sqcup \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2 \rrbracket,\end{aligned}$$

and

$$\begin{aligned}&\llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1 \rrbracket \sqcup \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2^* \rrbracket) \subseteq \\ &\qquad \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_1 \rrbracket \sqcup \llbracket \lambda\langle \nu_1 \cdots \nu_k \rangle \,.\, \varphi_2 \rrbracket.\end{aligned}$$

The result follows immediately. $\square$

13.8 Proposition For all formulas $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots \in$ LAL, we have $seq([\![\mathcal{F}_0]\!], [\![\mathcal{F}_1]\!], [\![\mathcal{F}_2]\!], \ldots) \subseteq [\![Seq(\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots)]\!]$.

Proof: We first notice the following two properties.

$$Seq(\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots) \equiv Before(\mathcal{F}_0, Seq(\mathcal{F}_1, \mathcal{F}_2, \ldots))$$

for all sequences of formulas $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots$, where $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots$ are either all liveness-sensitive, or all liveness-insensitive. Also, for all indexed families $F_0, F_1, F_2, \ldots$, the following holds.

$$seq(F_0, F_1, F_2, \ldots) = \widehat{before}(F_0, seq(F_1, F_2, \ldots))$$

The proposition is then proved by induction using Proposition 13.5. □

13.9 Proposition For all formulas $\mathcal{F}, \mathcal{F}' \in$ LAL, we have $\mathcal{F} \vdash \mathcal{F}'$ is a theorem in $\mathcal{L}^*$ only if $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$.

Proof: Theorems of the form $\mathcal{F} \vdash \mathcal{F}'$ can be obtained only using the axioms in Figure 13.2, and the modus ponens inference rule. As a result, once we have proved that the axioms have the property stated in the proposition, and that the inference rule preserves the property, it is immediate to produce an inductive proof. We shall limit ourselves to simply proving the that the proposition holds for the axioms and that the inference rule preserves it. For axioms (LAL1), (LAL2) and (LAL3), simply using the definition of the interpretation for LAL formulas and comparing the two indexed families will yield the result. In the case of the inference rule, we note that if $\mathcal{F} \vdash \mathcal{F}'$ and $\mathcal{F}' \vdash \mathcal{F}''$ are theorems, and $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ and $[\![\mathcal{F}']\!] \subseteq [\![\mathcal{F}'']\!]$, due to the transitivity of the $\subseteq$ relation, we have that $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}'']\!]$. □

14.1 Proposition Let $\mathcal{K}$ be a symbolic configuration over a well-defined language $\mathcal{L}$. Then $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$.

Proof: The proof is by induction on the structure of $\mathcal{K}$. Assume first that $\mathcal{K} = \langle l_s, \mathcal{F}_s\rangle \texttt{ skip } \langle l_f, \mathcal{F}_f\rangle$. We have that $[\![\mathcal{T}(\mathcal{K})]\!] = \langle l_s, [\![\mathcal{F}_s]\!]\rangle \texttt{ skip } \langle l_f, [\![\mathcal{F}_s]\!]\rangle = \widehat{T}([\![\mathcal{K}]\!])$.

Next, assume that $\mathcal{K} = \langle l_s, \mathcal{F}_s\rangle\, x := E\, \langle l_f, \mathcal{F}_f\rangle$. We have that
$[\![\mathcal{T}(\mathcal{K})]\!] = \langle l_s, [\![\mathcal{F}_s]\!]\rangle\, x := E\, \langle l_f, \widehat{assign}(x, E, [\![\mathcal{F}_s]\!])\rangle$, and
$\widehat{T}([\![\mathcal{K}]\!]) = \langle l_s, [\![\mathcal{F}_s]\!]\rangle\, x := E\, \langle l_f, [\![Assign(x, E, \mathcal{F}_s)]\!]\rangle$. Since $\mathcal{L}$ is well defined, it follows that $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$.

Assume now that $\mathcal{K} = \mathcal{K}_1 ; \mathcal{K}_2$. Obviously, $[\![\mathcal{K}]\!] = [\![\mathcal{K}_1]\!] ; [\![\mathcal{K}_2]\!]$. Using the induction hypothesis, we have that $\widehat{T}([\![\mathcal{K}_1]\!]) \subseteq [\![\mathcal{T}(\mathcal{K}_1)]\!]$ and $\widehat{T}([\![\mathcal{K}_2]\!]) \subseteq [\![\mathcal{T}(\mathcal{K}_2)]\!]$. We also have that $\widehat{T}([\![\mathcal{K}]\!]) = \widehat{T}([\![\mathcal{K}_1]\!]) \fatsemi \widehat{T}([\![\mathcal{K}_2]\!])$ and $[\![\mathcal{T}(\mathcal{K})]\!] = [\![\mathcal{T}(\mathcal{K}_2) \fatsemi \mathcal{T}(\mathcal{K}_2)]\!]$. From all these conditions it follows that $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$.

Next, assume that $\mathcal{K} = \langle l_s, \mathcal{F}_s\rangle \texttt{ if } C \texttt{ then } \mathcal{K}_c \texttt{ else } \mathcal{K}_a \texttt{ endif } \langle l_f, \mathcal{F}_f\rangle$. We have that

$$
\begin{aligned}
\widehat{T}([\![\mathcal{K}]\!]) = \;& \langle l_s, [\![\mathcal{F}_s]\!]\rangle \\
& \quad \texttt{if } C \;\; \texttt{then } \langle first(\mathcal{K}_c), \widehat{filter}(C, [\![\mathcal{F}_s]\!])\rangle \fatsemi \widehat{T}([\![\mathcal{K}_c]\!]) \\
& \qquad\quad \texttt{else } \langle first(\mathcal{K}_a), \widehat{filter}(\neg C, [\![\mathcal{F}_s]\!])\rangle \fatsemi \widehat{T}([\![\mathcal{K}_a]\!]) \\
& \quad \texttt{endif} \\
& \langle l_f, [\![\mathcal{K}_c]\!]|_{last(\mathcal{K}_c)} \;\widehat{\cup}\; [\![\mathcal{K}_a]\!]|_{last(\mathcal{K}_a)}\rangle,
\end{aligned}
$$

and

$$
\begin{aligned}
[\![\mathcal{T}(\mathcal{K})]\!] = \;& \langle l_s, [\![\mathcal{F}_s]\!]\rangle \\
& \quad \texttt{if } C \;\; \texttt{then } \langle first(\mathcal{K}_c), [\![Filter(C, \mathcal{F}_s)]\!]\rangle \fatsemi [\![\mathcal{T}(\mathcal{K}_c)]\!] \\
& \qquad\quad \texttt{else } \langle first(\mathcal{K}_a), [\![Filter(\neg C, \mathcal{F}_s)]\!]\rangle \fatsemi [\![\mathcal{T}(\mathcal{K}_a)]\!] \\
& \quad \texttt{endif} \\
& \langle l_f, [\![\mathcal{K}_c|_{last(\mathcal{K}_c)} \sqcup \mathcal{K}_a|_{last(\mathcal{K}_a)}]\!]\rangle.
\end{aligned}
$$

According to the induction hypothesis, we have $\widehat{T}([\![\mathcal{K}_c]\!]) \subseteq [\![\mathcal{T}(\mathcal{K}_c)]\!]$, and $\widehat{T}([\![\mathcal{K}_a]\!]) \subseteq [\![\mathcal{T}(\mathcal{K}_a)]\!]$. Since $\mathcal{L}$ is well defined, it follows that $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$.

Finally, assume that $\mathcal{K} = \langle l_s, \mathcal{F}_s\rangle$ `while` C `do` $\mathcal{K}_b$ `endwhile` $\langle l_f, \mathcal{F}_f\rangle$. We have that

$$\begin{aligned}\widehat{T}([\![\mathcal{K}]\!]) = \;& \langle l_s, [\![\mathcal{F}_s]\!]\rangle \\ & \texttt{while } C \texttt{ do} \\ & \quad \langle \mathit{first}(\mathcal{K}_b), \widehat{\mathit{filter}}(C, \widehat{\mathit{before}}([\![\mathcal{F}_s]\!], [\![\mathcal{K}_b]\!]|_{\mathit{last}(\mathcal{K}_b)}))\rangle \fatsemi \widehat{T}([\![\mathcal{K}_b]\!]) \\ & \texttt{endwhile} \\ & \langle l_f, \widehat{\mathit{filter}}(C, [\![\mathcal{F}_s]\!] \;\widehat{\cup}\; \mathit{collect}[\![\mathcal{K}_b]\!]|_{\mathit{last}(\mathcal{K}_b)}))\rangle,\end{aligned}$$

and

$$\begin{aligned}[\![\mathcal{T}(\mathcal{K})]\!] = \;& \langle l_s, [\![\mathcal{F}_s]\!]\rangle \\ & \texttt{while } C \texttt{ do} \\ & \quad \langle \mathit{first}(\mathcal{K}_c), [\![\mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, \mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))]\!]\rangle \fatsemi [\![\mathcal{T}(\mathcal{K}_b)]\!] \\ & \texttt{endwhile} \\ & \langle l_f, [\![\mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)})]\!]\rangle.\end{aligned}$$

According to the induction hypothesis, we have $\widehat{T}([\![\mathcal{K}_b]\!]) \subseteq [\![\mathcal{T}(\mathcal{K}_b)]\!]$. Since $\mathcal{L}$ is well defined, it follows that $\widehat{T}([\![\mathcal{K}]\!]) \subseteq [\![\mathcal{T}(\mathcal{K})]\!]$. $\square$

15.5 Proposition Let $\mathcal{K}$ be a symbolic configuration whose annotations contain *assert* constructs whose subscripts may be labels not necessarily in $\mathit{labels}(\mathcal{K})$. Let l and Λ be a label, and a set of labels, respectively, not necessarily from $\mathit{labels}(\mathcal{K})$, with $l \notin \Lambda$. If $\mathit{assume}_{\Lambda\cup\{l\}}(\mathcal{T}(\mathcal{K})|_{l'}) \vdash \mathit{assume}_{\{l\}}(\mathcal{K}|_{l'})$ holds for some label $l' \in \mathit{labels}(\mathcal{K})$, then $\mathit{assume}_{\Lambda\cup\{l\}}(\mathcal{T}((\mathcal{T}\sqcap\mathcal{I})(\mathcal{K}))|_{l'}) \vdash \mathit{assume}_{\{l\}}((\mathcal{T}\sqcap\mathcal{I})(\mathcal{K})|_{l'})$ holds.

Proof: The proof is by induction on the structure of $\mathcal{K}$. First, assume that $\mathcal{K}$ is the configuration $\langle l_s, \mathcal{F}_s\rangle$ `skip` $\langle l_f, \mathcal{F}_f\rangle$. Then, we have

$$\begin{aligned}\mathcal{T}(\mathcal{K}) &= \langle l_s, \mathcal{F}_s\rangle \texttt{ skip } \langle l_f, \mathcal{F}_s\rangle \\ (\mathcal{T}\sqcap\mathcal{I})(\mathcal{K}) &= \langle l_s, \mathcal{F}_s\rangle \texttt{ skip } \langle l_f, \mathcal{F}_s \sqcap \mathcal{F}_f\rangle \\ \mathcal{T}((\mathcal{T}\sqcap\mathcal{I})(\mathcal{K})) &= \langle l_s, \mathcal{F}_s\rangle \texttt{ skip } \langle l_f, \mathcal{F}_s\rangle\end{aligned}$$

If $l = l_s$, the proof is immediate. If $l = l_f$, the the proof follows from the fact that $assume_{\Lambda\cup\{l\}}(\mathcal{F}_s) \vdash assume_{\{l\}}(\mathcal{F}_f)$ holds implies that $assume_{\Lambda\cup\{l\}}(\mathcal{F}_s) \vdash assume_{\{l\}}(\mathcal{F}_s \sqcap \mathcal{F}_f)$. Assume now that $\mathcal{K}$ is the configuration $\langle l_s, \mathcal{F}_s\rangle\, x := E\, \langle l_f, \mathcal{F}_f\rangle$. Then, we have

$$\begin{aligned}
\mathcal{T}(\mathcal{K}) &= \langle l_s, \mathcal{F}_s\rangle\, x := E\, \langle l_f, Assign(x, E, \mathcal{F}_s)\rangle \\
(\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}) &= \langle l_s, \mathcal{F}_s\rangle\, x := E\, \langle l_f, \mathcal{F}_s \sqcap Assign(x, E, \mathcal{F}_s)\rangle \\
\mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K})) &= \langle l_s, \mathcal{F}_s\rangle\, x := E\, \langle l_f, Assign(x, E, \mathcal{F}_s)\rangle
\end{aligned}$$

If $l = l_s$, the proof is immediate. If $l = l_f$, the the proof follows from the fact that $assume_{\Lambda\cup\{l\}}(Assign(x, E, \mathcal{F}_s)) \vdash assume_{\{l\}}(\mathcal{F}_f)$ holds implies that $assume_{\Lambda\cup\{l\}}(Assign(x, E, \mathcal{F}_s)) \vdash assume_{\{l\}}(Assign(x, E, \mathcal{F}_s) \sqcap \mathcal{F}_f)$. Assume now that $\mathcal{K}$ is the configuration $\mathcal{K}_1 \fatsemi \mathcal{K}_2$. Then, we have

$$\begin{aligned}
\mathcal{T}(\mathcal{K}) &= \mathcal{T}(\mathcal{K}_1) \fatsemi \mathcal{T}(\mathcal{K}_2) \\
(\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}) &= (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_1) \fatsemi (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_2) \\
\mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K})) &= \mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_1)) \fatsemi \mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_2))
\end{aligned}$$

The proof follows by direct application of the induction hypothesis. Assume now that $\mathcal{K}$ is the configuration $\langle l_s, \mathcal{F}_s\rangle$ `if` C `then` $\mathcal{K}_c$ `else` $\mathcal{K}_a$ `endif` $\langle l_f, \mathcal{F}_f\rangle$. Then, we have

$$\begin{aligned}
\mathcal{T}(\mathcal{K}) = \quad & \langle l_s, \mathcal{F}_s\rangle \\
& \texttt{if } C \texttt{ then } \langle first(\mathcal{K}_c), Filter(C, \mathcal{F}_s)\rangle \fatsemi \mathcal{T}(\mathcal{K}_c) \\
& \qquad \texttt{else } \langle first(\mathcal{K}_a), Filter(\neg C, \mathcal{F}_s)\rangle \fatsemi \mathcal{T}(\mathcal{K}_a) \\
& \texttt{endif} \\
& \langle l_f, \mathcal{K}_c|_{last(\mathcal{K}_c)} \sqcup K_a|_{last(\mathcal{K}_a)}\rangle
\end{aligned}$$

$$
\begin{array}{lcl}
(\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}) & = & \langle l_s, \mathcal{F}_s \rangle \\
& & \quad \texttt{if } C \ \ \texttt{then } \langle \mathit{first}(\mathcal{K}_c), \mathcal{K}_c|_{\mathit{first}(\mathcal{K}_c)} \sqcap \mathit{Filter}(C, \mathcal{F}_s) \rangle \\
& & \qquad \fatsemi (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_c) \\
& & \quad \texttt{else } \langle \mathit{first}(\mathcal{K}_a), \mathcal{K}_a|_{\mathit{first}(\mathcal{K}_a)} \sqcap \mathit{Filter}(\neg C, \mathcal{F}_s) \rangle \\
& & \qquad \fatsemi (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_a) \\
& & \quad \texttt{endif} \\
& & \langle l_f, \mathcal{F}_f \sqcap (\mathcal{K}_c|_{\mathit{last}(\mathcal{K}_c)} \sqcup K_a|_{\mathit{last}(\mathcal{K}_a)}) \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K})) & = & \langle l_s, \mathcal{F}_s \rangle \\
& & \quad \texttt{if } C \ \ \texttt{then } \langle \mathit{first}(\mathcal{K}_c), \mathit{Filter}(C, \mathcal{F}_s) \rangle \fatsemi \\
& & \qquad \mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_c)) \\
& & \quad \texttt{else } \langle \mathit{first}(\mathcal{K}_a), \mathit{Filter}(\neg C, \mathcal{F}_s) \rangle \fatsemi \\
& & \qquad \mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_a)) \\
& & \quad \texttt{endif} \\
& & \langle l_f, (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_c)|_{\mathit{last}(\mathcal{K}_c)} \sqcup (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_a)|_{\mathit{last}(\mathcal{K}_a)} \rangle
\end{array}
$$

We now have six cases.

$l = l_s$:

The proof here is immediate.

$l = \mathit{first}(\mathcal{K}_c)$:

The proof follows from the fact that

$$\mathit{assume}_{\Lambda \cup \{l\}}(\mathit{Filter}(C, \mathcal{F}_s)) \vdash \mathit{assume}_{\{l\}}(\mathcal{K}_c|_{\mathit{first}(\mathcal{K}_c)})$$

holding implies

$$\mathit{assume}_{\Lambda \cup \{l\}}(\mathit{Filter}(C, \mathcal{F}_s)) \vdash \mathit{assume}_{\{l\}}(\mathit{Filter}(C, \mathcal{F}_s) \sqcap \mathcal{K}_c|_{\mathit{first}(\mathcal{K}_c)})$$

holds.

$l = \mathit{first}(\mathcal{K}_a)$:

The proof follows from the fact that

$$\mathit{assume}_{\Lambda \cup \{l\}}(\mathit{Filter}(\neg C, \mathcal{F}_s)) \vdash \mathit{assume}_{\{l\}}(\mathcal{K}_a|_{\mathit{first}(\mathcal{K}_a)})$$

holding implies

$$\mathit{assume}_{\Lambda \cup \{l\}}(\mathit{Filter}(\neg C, \mathcal{F}_s)) \vdash \mathit{assume}_{\{l\}}(\mathit{Filter}(\neg C, \mathcal{F}_s) \sqcap \mathcal{K}_a|_{\mathit{first}(\mathcal{K}_a)})$$

holds.

$l \in \mathit{labels}(\mathcal{K}_c) \backslash \{\mathit{first}(\mathcal{K}_c)\}$:

The proof follows from the induction hypothesis.

$l \in \mathit{labels}(\mathcal{K}_a) \backslash \{\mathit{first}(\mathcal{K}_a)\}$:

The proof follows from the induction hypothesis.

$l = l_f$:

The proof follows from the fact that

$$\mathit{assume}_{\Lambda \cup \{l\}}(\mathcal{K}_c|_{\mathit{last}(\mathcal{K}_c)} \sqcup K_a|_{\mathit{last}(\mathcal{K}_a)}) \vdash \mathit{assume}_{\{l\}}(l_f)$$

holding implies that

$$\mathit{assume}_{\Lambda \cup \{l\}}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_c)|_{\mathit{last}(\mathcal{K}_c)} \sqcup (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_a)|_{\mathit{last}(\mathcal{K}_a)}) \vdash \mathit{assume}_{\{l\}}(l_f)$$

holds.

Assume now that $\mathcal{K}$ is the configuration $\langle l_s, \mathcal{F}_s\rangle$ `while` C `do` $\mathcal{K}_b$ `endwhile` $\langle l_f, \mathcal{F}_f\rangle$.

Then, we have

$$
\begin{array}{rcl}
\mathcal{T}(\mathcal{K}) & = & \langle l_s, \mathcal{F}_s\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \quad\quad \langle \mathit{first}(\mathcal{K}_b), \mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, K_b|_{\mathit{last}(K_b)}))\rangle \\
& & \quad\quad\quad \fatsemi \mathcal{T}(\mathcal{K}_b) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle l_f, \mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))\rangle
\end{array}
$$

$$
\begin{array}{rcl}
(\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}) & = & \langle l_s, \mathcal{F}_s\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \quad\quad \langle \mathit{first}(\mathcal{K}_b), K_b|_{\mathit{first}(\mathcal{K}_b)} \sqcap \mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, \mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))\rangle \\
& & \quad\quad\quad \fatsemi (\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_b) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle l_f, \mathcal{F}_f \sqcap \mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K})) & = & \langle l_s, \mathcal{F}_s\rangle \\
& & \quad \texttt{while } C \texttt{ do} \\
& & \quad\quad \langle \mathit{first}(\mathcal{K}_b), \mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, K_b|_{\mathit{last}(K_b)}))\rangle \\
& & \quad\quad\quad \fatsemi \mathcal{T}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_b)) \\
& & \quad \texttt{endwhile} \\
& & \quad \langle l_f, \mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_b)|_{\mathit{last}(\mathcal{K}_b)}))\rangle
\end{array}
$$

We now have four cases

$l = l_s$:

The proof here is immediate.

$l = \mathit{first}(\mathcal{K}_b)$:

The proof follows from the fact that

$$\mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, \mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)})) \vdash \mathcal{K}_b|_{\mathit{first}(\mathcal{K}_b)}$$

holding implies that

$$\mathit{Filter}(C, \mathit{Before}(\mathcal{F}_s, \mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)})) \vdash \mathcal{K}_b|_{\mathit{first}(\mathcal{K}_b)} \sqcap \mathit{Before}(\mathcal{F}_s, \mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))$$

holds.

$l \in \mathit{labels}(\mathcal{K}_b) \backslash \{\mathit{first}(\mathcal{K}_b)\}$:

The proof follows from the induction hypothesis.

$l = l_f$:

The proof follows from the fact that

$$\mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)})) \vdash \mathcal{F}_f$$

holding implies that

$$\mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}((\mathcal{T} \sqcap \mathcal{I})(\mathcal{K}_b)|_{\mathit{last}(\mathcal{K}_b)})) \vdash \mathcal{F}_f \sqcap \mathit{Filter}(\neg C, \mathcal{F}_s \sqcup \mathit{Collect}(\mathcal{K}_b|_{\mathit{last}(\mathcal{K}_b)}))$$

holds.

□