Semantics of VLSI Synthesis

Thesis by
Marcel René van der Goot

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California
1995

(Submitted 26 May 1995)

Acknowledgements

Many people have contributed directly or indirectly to my work and this thesis. A few are listed below. My thanks, and apologies, to anyone I forgot to mention.

First of all, I would like to thank my research adviser, Alain Martin, for accepting me as a student, for letting me stay all these years, and for creating an atmosphere where I've always felt free to speak my mind.

I would like to thank the members of my thesis defense committee, Alain, Mani Chandy, Beverly Sanders, Yaser Abu-Mostafa, and Peter Hofstee, for agreeing to meet on short notice, and for the time spent reviewing this text.

My thanks to Alain and the past and current members of his research group, especially Steve Burns, Pieter Hazewindus, Tony Lee, Drazen Borkovic, and José Tierno, for developing the VLSI synthesis method which provides the background for this thesis, and for many useful discussions.

Christian Nielsen deserves thanks for convincing me that a semantics for the VLSI synthesis method would indeed be a good thesis topic. I thank Ralph Back for introducing me to formal operational semantics: this opened up a whole new world for me.

José, Peter, and Jessie Xu read early drafts of the first few chapters; their comments led to great improvements in the presentation. Rajit Manohar, Peter, and Alain read the complete final draft, finding several errors and mistakes in the process. Any remaining errors can only be blamed on me.

This research was sponsored by the Advanced Research Projects Agency; their support is greatfully acknowledged.

My eight years at Caltech have been pleasant ones thanks to many friends. Special thanks go to Jan van de Snepscheut, my undergraduate adviser, who suggested a graduate study at Caltech; to Arlene DesJardins, who helped me settle in here in Pasadena; to John Ngai, who helped me through the first few years; to Mike Pertel, who helped explore Los Angeles, introduced me to skiing, and set a great example of courage; to Drazen and José, who never tired of explaining to me how things work; to Tony, who is a great companion both on and off campus; and to Peter, who was always willing to listen, and often gave useful advice. Thanks to all my other friends as well.

I thank my parents and my family for their love, their never wavering support, and the many pleasant vacations spent together. I thank my late sister, Ingrid, for the many things she taught me about life. I especially thank my wife, Claire, for making my life a happy one, and for enduring my much longer than expected tenure as a graduate student. Finally, I thank my daughter, Denise, for never failing to brighten my day.

To my parents, and to Ingrid, Claire, and Denise

Abstract

We develop a new form of formal operational semantics, suitable for concurrent programming languages. The semantics directly supports sequential and parallel composition, rendezvous synchronization, shared variables, and non-determinism. Based on an abstract notion of program execution, a refinement relation is defined. We show how the refinement relation can be used to prove that one program implements another.

We use the operational semantics as a semantic framework for a synthesis method for asynchronous VLSI circuits. We define the semantics of the programming notations that are used, and use the refinement relation to prove the correctness of the program transformations that form the basis of the synthesis method. Among other transformations, we proof the correctness of the replacement of atomic synchronization actions by handshake protocols, and the transformation of a sequence of actions into a network of concurrently executing gates.

Contents

Chapter 1
Introduction

For more than a decade, the Caltech research group led by A.J. Martin has been designing VLSI circuits using a systematic synthesis method. This method is refinement-based: a correct but potentially inefficient program (the specification) is repeatedly transformed until a design is obtained that is both efficient and in a form that directly describes a circuit (a gate description). Hence, circuit design using this method resembles program design, with a compilation stage that transforms the programming notation to a circuit description. If each of the transformations is semantics-preserving, then the correctness of the initial design implies the correctness of the eventual implementation. Although many working circuits have been designed using this method [13, 15, 18, 25], there was no uniform formal framework for proving that the transformations are indeed semantics-preserving. In this text, we describe such a framework, and use it to prove the correctness of most standard transformations used in the design method.

1.1 Semantics

A transformation maps a program to a different program. In order to prove that a transformation preserves a program's semantics, this semantics must first be defined. A semantics assigns a meaning to programs; for instance, this can be done by mapping programs to objects of which the meaning is already known. We are interested in a formal semantics for our programs. *Formal* means that the semantic description uses precise formulas. Our desire for formality is based on our desire for precision: it is nearly impossible to decide whether a proof is precise and, especially, complete, if there is no precise and complete description of the proof requirements.

In terms of choosing a suitable program semantics, the programs we are interested in have several features that pose problems. In particular, the semantics must be able to define

- concurrency;
- rendezvous synchronization;
- shared variables; and
- non-terminating computations.

Many methods for defining semantics only support a subset of these features. For instance, traditional predicate transformer semantics [5] does not directly support concurrency; an extension like action systems [1] can be used to define concurrency, but cannot express synchronization; trace theory [23] supports synchronization, but no shared variables. Since we want to use the semantics as part of an existing method, rather than to develop a new language, it is not useful to omit any of these problem features from consideration.

Support of the necessary language features is not the only issue when choosing a semantics. Although no formal framework for the VLSI design method existed, there are of course informal arguments justifying the transformations. It is preferable if formal proofs can be created by formalizing these existing arguments (possible making them more precise and complete), rather than by following an entirely different line of reasoning. For instance, process algebra [2, 9, 16] is a popular formal method for proofs (and derivations) of VLSI circuits [6, 10]. But proofs and derivations in such algebras seem very far removed from our usual programming and design methods. This is not unique to VLSI design. Much work in programming language semantics and formal derivations shares this property that the formal method is entirely different from the informal method that is used in practice, even if that informal method is systematic and precise. As a result, formal methods are only rarely used in practical applications. In VLSI design, where the use of systematic design and formal methods is less common than in programming, this problem seems likely to be worse. Since we would like VLSI designers to use our semantics and proofs, we prefer a semantics that uses the notations and concepts of existing practical design methods, and that allows proofs to mirror the lines of informal reasoning used to justify transformations.

Because of this preference, we decided to use *operational semantics*. A semantics is operational if it defines an (abstract) notion of *program execution*, and defines the meaning of programs through the result of such executions. Note that whether a semantics is operational or not is independent of whether it is formal or not: the latter attribute refers only to the way the semantics is presented. Informal reasoning is often based on operational arguments, so that there is hope that proofs in an operational semantics can mirror the informal reasoning.

Our goal is not just to assign meaning to programs: we want to prove that transformations preserve these meanings. If a program S is transformed into another program T, preserving the meaning means that T *implements* (or *refines*) S. Hence, our purpose in defining a semantics is to define what it means for one program to implement another. The standard method for describing operational semantics, [19, 20], does not quite do this: it defines a notion of program execution, but does not really define what the result of such execution is. Therefore, it is hard to use the result of execution to compare programs. Other methods, usually called *refinement calculi*, do define an implementation relation, but not one based on operational semantics.

Based on these considerations, we decided to develop our own method of operational semantics, designed to support all the language features needed, and with the express purpose of defining an implementation relation. The semantics gives precise definitions of such familiar concepts as programs, program execution, and computations, and uses these concepts to define an implementation relation.

1.2 VLSI

Abstraction is an important tool to build correct systems of increasing complexity. Abstraction is mainly a method of separation of concerns: it allows part of a design to be done without concern for every issue that is relevant in the final implementation. For instance, high-level programming languages allow one to write correct programs largely without concern for execution time and memory usage. Abstraction not only enables designs of larger complexity, it also increases the portability of such designs.

These arguments are as true for VLSI design as for program design. Since the complexity of VLSI circuits is on the same scale as the complexity of large programs, it makes sense to employ similar techniques to manage that complexity. In particular, it is useful if a circuit can be described as a program in a more or less conventional programming language. This requires that many physical properties of circuits are abstracted from the correctness concerns. Examples of such physical properties are layout, voltages of signals, switching speed, and energy usage. However, the implementation method must support such abstraction. For instance, although it may be possible to show the "logical correctness" of a design by simply ignoring timing issues, if the actual implementation critically depends on timing properties for correctness, such a method is of limited use for correctness proofs: at some point, the abstract design must match with the physical reality [21].

Although most VLSI design methods abstract from layout properties, not all allow abstraction from timing issues. In particular, timing plays an important role in traditional clocked designs (also called synchronous designs). Asynchronous design methods, on the other hand, allow a large degree of abstraction from timing issues: rather than using a clock, they rely on request and acknowledgement signals to enforce the proper order of actions. In the past, few designs were asynchronous, mostly because no effective method for designing large asynchronous circuits was known. As mentioned, by building many successful circuits, the Martin synthesis method has been shown to be an effective method for asynchronous circuit design. It overcomes many traditional problems, such as hazards and races, by using the proper abstraction and implementation methods. By abstraction of timing issues and by the use of a systematic refinement method, it reduces the complexity of the design process, thus enabling faster design of larger circuits than more traditional methods. Because of the large degree of abstraction, the method is a suitable candidate for formalization. Although there is no inherent reason why timing issues cannot be formalized, the bulk of the work on programming semantics and formal derivations does not take time into account, suggesting that there is still a long way to go before correctness proofs for synchronous systems can be given.

The existence of a systematic stepwise refinement method, the larger degree of abstraction, and, with this text, the existence of a formal semantic framework, are not the only reasons to prefer asynchronous design over synchronous systems. In particular, the speed of an asynchronous circuit is determined by the average case delay, rather than by the worst case delay as in a synchronous circuit. Hence, asynchronous circuits have the potential to be faster than their synchronous counterparts. Furthermore, only the parts of an asynchronous circuit that contribute to a computation are switching, whereas in a synchronous circuit all parts that are reached by the clock signal switch, regardless of whether that contributes to the result. Hence, asynchronous circuits also have the potential for lower energy usage than synchronous circuits. It is important to realize that abstraction does not mean that issues like speed and energy cannot be taken into account: rather, it means that each issue can be considered in separation. For instance, each transformation applied to a design must be correct (i.e., semantics-preserving), but the choice of which correct transformation to apply is normally guided by efficiency concerns. In particular, [3] and [11] describe how optimization for speed can be integrated with the design process, whereas [26] introduces a method to make early design decisions guided by energy consumption concerns. Finally, the increased abstraction increases the portability of designs. For instance, the design of the Caltech Asynchronous Microprocessor was implemented in both CMOS [15] and GaAs [25], with virtually identical gate descriptions.

The Martin synthesis method, on which this text is based, starts a design with a program, often sequential, written in a CSP-like notation. The CSP notation supports the concurrent operation of sequential processes, which communicate and synchronize using rendezvous communication over channels. This initial program is transformed several times, until a program with a large degree of concurrency, and of the proper form, is obtained. The transformations that are applied during this part of the design often depend on the particular program. The transformations are intended to increase efficiency (e.g., by introducing pipelining) and to obtain a program of the proper form for the next design stage. This part of the design corresponds to what is usually called program refinement. The following figure illustrates the design process, with each triangle corresponding to one or more transformations. The program refinement stage corresponds to the horizontal (CSP $\triangleright$ CSP) transformations.

$$\begin{array}{ccccccc} \mathrm{CSP} & \triangleright & \mathrm{CSP} & \triangleright & \ldots & \triangleright & \mathrm{CSP} \\ &&&&&& \triangledown \\ &&&&&& \mathrm{HE} \\ &&&&&& \triangledown \\ &&&&&& \mathrm{PRS} \\ &&&&&& \triangledown \\ &&&&&& \mathrm{CMOS} \end{array}$$

When an appropriate CSP program has been achieved, it is transformed to a different notation, call Handshaking Expansion (HSE). The main purpose of this transformation is to replace all rendezvous synchronizations by protocols using shared variables. Eventually, the HSE program is transformed to a gate-level description in a notation called Production Rule Sets (PRS). The gate-level description is then mapped to the implementation medium, for instance, CMOS. The transformations used in this stage of the design, depicted vertically in the above figure, come from a relatively small collection and are quite systematic. Therefore, this stage of the design closely resembles compilation (silicon compilation, in this case).

Because there is considerable effort involved in proving each transformation, we restrict ourselves to proving the correctness of the more systematic transformations used during the compilation phase. (I.e., the transformations that transform a CSP program into a PRS program.) That does not mean that the formal framework cannot be used to prove the 'horizontal transformations.' But due to the large variety of those transformations, this may be better done as part of each individual design. We should point out that the formal semantic framework is used to prove the correctness of the transformations, not to guide in chosing which transformation to apply. The power of abstraction is exactly that these concerns can be separated.

Some of the transformations that are used in the Martin synthesis method have been proven by Smith and Zwarico in [22]. Their approach is based on process algebra, where the rules of the algebra are related to an operational semantics. Although their goals are similar to ours, the use of process algebra makes the proofs quite different, and, in our opinion, less closely related to the way the design method is usually applied (which, in turn, makes the proofs harder to understand). Another important difference is that our implementation relation is based on arbitrary environments, whereas their definition requires the use of special testing environments.

1.3 Outline

This text consists of two parts. The first half gives a formal description of the semantic framework, and uses it to define the meaning of the CSP and HSE programming notations (the PRS programming notation is defined later, using the same framework). As explained, the purpose of the semantics is to define an implementation relation. This part of the text is quite independent of any VLSI design method, except that the choice of features for the semantics (concurrency, rendezvous synchronization, etc.) is guided by the features of the languages we want to use it for. Because we develop a new form of semantics, no knowledge of other semantics is needed, but familiarity with formal methods will be an advantage.

The second part uses the implementation relation to prove transformations used in the Martin synthesis method. Since the synthesis method uses rather standard programming notations, and has abstracted from physical circuit properties, no knowledge of VLSI technology is needed to understand this part of the text. However, familiarity with the synthesis method will lead to a better understanding of the relevance of the proved transformations. In this text we restrict ourselves to mentioning the approximate purpose of the transformations; for a description of the complete synthesis method, see [14].

- Chapter 2 starts the description of the semantics, by formally defining what programs and computations are.
- Chapter 3 continues the description by defining how programs are executed, thus establishing a relation between programs and computations. The chapter ends with the definition of the implementation relation.
- Chapter 4 provides the transition between the semantics and the proofs of transformations. Here we introduce techniques for proving transformations with the formal semantics, and illustrates this by proving several general transformations.
- Chapter 5 proves the correctness of the transformations used to transform a CSP program to a HSE program. In particular, we concentrate on replacing rendezvous synchronization by handshake protocols using shared variables.
- Chapter 6 defines the semantics of PRS programs, and proves under which conditions a PRS program implements a HSE program.
- Chapter 7 summarizes the results and lists some directions for future research.
- Appendix A explains the way we use abstract data types in our semantics. It shows how the use of abstract data types in Chapter 2 can be made more rigorous.

On a final note, recall that one of our goals is to provide a semantic framework that can be easily understood, uses familiar notations, and has a clear relation to the existing design method. To further this goal, we often use conventional names for formal concepts, such as 'computation,' 'execution,' and 'environment'; use of easily understood terms does not imply informality or imprecision. The reader should be careful to base proofs on the formal definitions, not on assumed properties of the informal name. We also have attempted to make the text easier to read by reducing the amount of formal notation. In general, whenever a notation does not seem to contribute to the understanding of a proof, it has been simplified, after first explaining the complete concept. For instance, a program corresponds to a tuple $\langle t, \mathcal{V}(t), \mathcal{I}(t)\rangle$. But since t, which describes the code, is by far the most important, we usually equate programs with just the first field of the tuple. Otherwise we would have to start each argument with a phrase of the form "let $T = \langle t, \mathcal{V}(t), \mathcal{I}(t)\rangle$ be a program," which does not contain essentially more information then "consider program t." Finally, we sometimes omit the braces around the elements of a set or bag. E.g., we write $\mathsf{ext}(\emptyset; t_1, t_2)$ instead of $\mathsf{ext}(\emptyset; \{t_1, t_2\})$ (see Chapter 2), and $T \cup t$ instead of $T \cup \{t\}$.

Chapter 2

Semantics: Computations and Programs

In this chapter and the next one we define our semantic method. As explained in Section 1.1, the purpose of the semantics is to define when a program implements another. In this chapter we describe how programs* and computations are represented in the semantics. Both concepts are represented by certain mathematical objects. However, rather than choosing existing mathematical data types (such as sets or lists), we define new ***abstract data types***. This allows us to tailor the data types to support exactly the language features we want. This is important, because if a formal semantics is given for a language that is already in use, it is necessary to convince oneself that the new formal description is equivalent to the existing descriptions. Naturally, this cannot be formally proven if the existing descriptions are not within a formal framework.

Convincing oneself of a semantic's correctness is easier if language features map directly to constructs of the semantics. For instance, in Chapter 5 we prove that rendezvous synchronization can be implemented using a handshake protocol with shared variables. Hence, a semantic framework that supports shared variables can also express synchronization, by defining it as a handshake protocol. However, it is unclear how one should then convince oneself that the semantics indeed defines synchronization. Likewise, in Chapter 6 we show how sequential programs are implemented in a concurrent language without sequencing, called PRS. The PRS language resembles Unity [4], hence it is tempting to use the Unity proof methods. However, since Unity does not directly support sequencing, we would have to use the methods of Chapter 6 to define sequential composition for the other notations that we use. This would again make it hard to show the correctness of the semantics. Therefore, we choose the semantics so that it can directly support the language features of interest to us.

In the last section of this chapter we define the semantics of standard programming constructs by mapping them to the earlier defined abstract data type for programs.

* In this text, the term *program* always refers to a program *part*; for instance, a single assignment, a loop, and a sequence of statements, are all examples of programs.

The use of abstract data types in semantics is not new. For instance, action semantics [17] uses ADTs for the same reason we do, namely that that way programs can be mapped to mathematical objects in a natural way. Apart from the use of ADTs, however, there is not much relation between action semantics and our semantics.

2.1 Data

It is often useful to separate the aspects of a program or programming language into *data* and *control*. With *data* we refer to the values of variables and expressions, as well as to components of the programming language that are directly related, such as operators, declarations, and assignments. With *control* we refer to the constructs that determine which operations are performed on the data and in which order, and to constructs that build larger programs out of smaller parts. Control includes composition methods such as sequential and parallel composition, as well as constructs such as loops and choices. Of course, there are interactions between data and control: for instance, choices (which are control constructs) are normally based on the value of data. Also, some variables, such as a program counter, may be more properly considered part of the control than part of the data. Nevertheless, the separation of concerns provided by distinguishing between data and control is a useful principle that one encounters in many areas of computer science: in software (algorithms and data structures), in hardware (control circuitry and datapaths), as well as in semantics. This section describes how data aspects are represented in our semantics; control is described in Section 2.3.

We call the set of program variables with their values the *state* of the program. During the execution of a program, the state changes. (Note that we do not include the 'current position' during execution of a program in the state, unless explicitly represented by a program variable.) A state then is simply a mapping from variables to values. We will not be particularly concerned with the types of variables, although in most cases we use only booleans. We assume that all expressions are proper, in the sense that they obey type restrictions and that they only involve variables that are in the domain of the state. We call the set of all states, or the *type* of states, '*State*'.

In addition to the standard function notation for states, we use the following notation to specify changes in states. Let σ be a state, u and v variables, and a an expression of the type of u. Then $\sigma[u \mapsto a]$ is also a state, defined by

$$\sigma[u \mapsto a](v) = \begin{cases} a & \text{if } v \text{ is } u \\ \sigma(v) & \text{if } v \text{ is not } u \end{cases}$$

and with domain $\mathcal{D}(\sigma[u \mapsto a]) = \mathcal{D}(\sigma) \cup \{u\}$,

EXAMPLE 2.1

Let u and v be integer variables. Assume σ is a state such that $\sigma(u) = 1$ and $\sigma(v) = 2$.

- Let $\tau = \sigma[u \mapsto 3]$. Then $\tau(u) = 3$ and $\tau(v) = \sigma(v) = 2$.
- Let $\rho = \sigma[w \mapsto 4]$. Then $\rho(w) = 4$, regardless of whether $\sigma(w)$ is even defined.

□

It is convenient to extend the domain of states from variables to expressions; this also serves as the semantics of expressions. For instance, the meaning of the '+' operator is defined by

$$\sigma(a_1 + a_2) = \sigma(a_1) + \sigma(a_2)$$

The other operators are defined in a similar way; since the method is obvious, we will not state those definitions. (Others sometimes use the notation '$\sigma[\![a]\!]$' instead of '$\sigma(a)$' when a is an expression.)

States can be used to define the effect of assignments. Let a be an expression of the type of variable u, such that $\sigma(a)$ is defined. If we use conventional Hoare triples, the effect of an assignment is

$$\{\,\sigma\,\}\ u := a\ \{\,\sigma[u \mapsto \sigma(a)]\,\}$$

Since a is an expression in terms of program variables, it must be evaluated with respect to a state, σ in this case; we cannot just write $\sigma[u \mapsto a]$.

A variant of the assignment is the multiple assignment: (let b be an expression of the type of variable v, such that $\sigma(b)$ is defined)

$$\{\,\sigma\,\}\ u, v := a, b\ \{\,\sigma[u \mapsto \sigma(a)][v \mapsto \sigma(b)]\,\}$$

Note that b is evaluated with respect to σ, not with respect to $\sigma[u \mapsto \sigma(a)]$. This means that a and b are evaluated before any assignment is performed. Programming notations that allow multiple assignments usually require that u and v are different variables, so that the order of the assignments is irrelevant.

In this text we often use boolean assignments $u\!\uparrow$ and $u\!\downarrow$, defined by

$$\{\,\sigma\,\}\ u\!\uparrow\ \{\,\sigma[u \mapsto \mathbf{true}]\,\}$$

$$\{\,\sigma\,\}\ u\!\downarrow\ \{\,\sigma[u \mapsto \mathbf{false}]\,\}$$

It follows that assignments can be seen as mappings from a state to a state, called *state transformers*. Our semantics therefore equates assignments with state transformers, which have type *State* $\to$ *State*. Consequently, we will often write $u := a$ for the mapping that changes σ to $\sigma[u \mapsto \sigma(a)]$, and likewise for the other assignments.

Note that state transformers can be specified independently of the domains of states they are applied to. However, if the state transformer applies its argument state to some variables, we will assume that those variables are in the state's domain. For instance, if $u := v + w$ is applied to σ, we assume that the domain of σ, $\mathcal{D}(\sigma)$, contains v and w, but not necessarily u; the domain of the resulting state $\sigma[u \mapsto \sigma(v + w)]$ *does* contain u.

We often encounter functions of type *State* $\rightarrow$ *Boolean*, which map states to truth values. We call such functions *guards* (in other contexts they are often called *predicates*). The guard that maps σ to $\sigma(e)$ is written as simply the expression e (hence, $e(\sigma) = \sigma(e)$). As with state transformers, if a guard is applied to state σ, we assume that $\mathcal{D}(\sigma)$ contains all variables that σ is applied to.

It is often useful to restrict the domain of a state to a given set of variables. This is called the *projection* of the state onto that set.

DEFINITION 2.2 *(Projection)*

If σ is a state and A a set of variables, then the projection of σ on A, written $\sigma\lfloor A$, is the state with domain $\mathcal{D}(\sigma) \cap A$ and $(\sigma\lfloor A)(u) = \sigma(u)$.

$\square$

A useful property of projection is that if $A \subseteq B$, then $\sigma\lfloor A = (\sigma\lfloor B)\lfloor A$.

We extend projection to sets of states by applying the projection to each element. If X and Y are sets of states, then $X \subseteq Y$ implies that $X\lfloor A \subseteq Y\lfloor A$.

2.2 Computations

We are concerned with the observable behavior of a program when it is executed. This behavior we call a *computation*. Hence, a computation is the result of program execution. In order to define computations, we must first decide which aspects of program execution are observable. For purpose of this text, we decide that only variables and their values are observable; hence, computations are built from states. This is not the only possible choice of observable behavior: in other contexts it may be appropriate to consider execution time, memory usage, or energy usage as relevant to the computation. We will ignore those quantities — in particular, we will not make timing assumptions. (We do, however, distinguish between finite and infinite time, i.e., between termination and non-termination.)

A second question is, who or what does the observing, and how is this done? In our case, the observations are made by an *environment* of the program. Environments, which are described in detail in Section 3.3, can be composed with programs to form new programs that can then be executed. Usually, there are restrictions on which environments can be composed with which programs. The exact ways in which program and environment can be composed are described in Section 3.3; here we only distinguish between two major classes, sequential and parallel composition. Environment and program interact in two possible ways: through data, i.e., by observing and changing the state, and through control, i.e., by affecting the way in which execution progresses.

If an environment is composed *sequentially* with a program, the execution will consist of partial execution of the environment, followed by execution of the program, followed by partial execution of the environment (and maybe a repetition of this process). Hence, the data interaction is simple: during the execution there are only two states that are shared between environment and program, namely the initial and final states of the program.

Therefore, the program's computation can be described by just listing those two states, as we did in the previous section with a Hoare triple. The program can also interact with the environment through control: if the program does not terminate, then the environment will never be executed once the program is started. (We ignore the control interactions between the part of the environment executed before the program is started and the program, because our focus is on the program's behavior rather than the environment's behavior.) If we want to describe a non-terminating computation with a Hoare triple, we can introduce a special state, '∞' say, to use as 'post condition' for such a computation. It is also possible that a program attempts to perform an illegal operation, such as division by zero. Since we are interested in program transformations, we will assume that the initial program is free of such errors. If needed, it can be handled by adding an extra 'error variable' to the state, or by introducing a special error state. In many theories of sequential programming, non-termination is considered equivalent to an error.

In summary, a sequential program can have the following types of interaction with its environment:

- Data interaction: initial and final state
- Control interaction: non-termination

We call a program that is intended to be composed sequentially with its environment a *sequential program*, even though the program may very well have internal parallelism.

EXAMPLE 2.3

- An interactive Pascal program is not a sequential program, since the environment (i.e., the user) is supposed to 'execute' simultaneously with the program.
- A program executed as a batch job is a sequential program (with respect to the user as environment), even if it is executed on a parallel computer, because it does not interact with the user while executing.

□

In this text we do not restrict ourselves to sequential composition, but also allow the environment to be composed *in parallel* with a program. By analogy with the term sequential program, a program that may be composed in parallel with its environment is called a *parallel program*. If the environment and program are composed in parallel, the environment is executed simultaneously with the execution of the program. As a result, the data interaction is more complex than for sequential programs, since the environment can potentially observe and change any state during the program's execution. Therefore, to describe the computation, we list all states that occur during the execution. Such a list of states is called a *trace*. The control interaction of parallel programs is also more complex. Since we still allow sequential composition, non-termination is one way of control interaction. However, now it is useful to distinguish two types of non-termination. The first occurs when a program keeps producing states forever; this results in an infinite computation and therefore in an infinite trace. Unlike the situation with sequential programs, for parallel programs this type of non-termination is not considered an error; in fact, in many situations it is the rule rather than the exception. The second type of non-termination occurs when the execution can only continue if a certain state occurs, and that state does in fact not occur. In particular, this happens if the program is waiting for the environment to change the state, while the environment is waiting for the program to do the same thing. This second form of non-termination is called *deadlock*. Deadlock does not lead to an infinite trace. To distinguish between termination and deadlock, we introduce a special deadlock state '$\perp$'; whenever a computation gets into a deadlock (in other words, the computation *blocks*), we add $\perp$ as last element of the trace.

Most parallel programming languages have some construct to synchronize between program and environment, which introduces an additional form of control interaction. Synchronization between program and environment does not change the state. But since synchronization requires a cooperative effort of program and environment, it can potentially fail. If program and environment attempt to synchronize in incompatible ways, a situation similar to the above deadlock arises: the execution cannot continue because a certain synchronization condition has not been established. This situation is also called deadlock, and is indicated by the same $\perp$ state in the trace.

Hence, a parallel program can have the following types of interaction with its environment.

- Data interaction: any state
- Control interaction: non-termination
    - infinite computation/trace
    - deadlock ($\perp$)
- Control interaction: synchronization
    - successful synchronization is not observable
    - unsuccessful synchronization leads to deadlock ($\perp$)

Next we give a formal definition of traces. This definition is in the form of an abstract data type (ADT). The ADT defines a set of objects with certain properties. Such a set of objects is called a *type*. For instance, the type of states is *State*, the set of all states; the type of real numbers is $\mathbb{R}$. Some types are constructed from one or more other types, such as a set of states. We denote the type 'set of states' by **set of** (*State*). To discuss properties of sets that do not depend on the particular type of elements of the set, we use **set of** (A), where A is an arbitrary type. Traces are also constructed from another, arbitrary, type; hence, below we define a type **trace of** (X). At the end of this section we will make a specific choice for X to represent computations. We define the ADT by first describing what the form is of elements of the ADT (i.e., we describe the constants and constructors), and then stating some axioms about these elements. More details about ADTs are given in Appendix A; in particular, that appendix lists some implicit assumptions we make to simplify the definition. It should be pointed out that the purpose of the ADT is to formally define traces, so that later we can refer to them; we have no intention of using the given axioms to prove lots of properties of traces. The most important part of the definition is axiom t5.

Let *Trace* = **trace of** (X). For any $x \in X$ and any $p, q, r \in$ *Trace*, the following axioms hold.

- Constants and constructors

  t1. $\emptyset \in \mathit{Trace}$
  $\infty \in \mathit{Trace}$
  $x \in \mathit{Trace}$

  t2. $p +\!\!+\, q \in \mathit{Trace}$

- Equivalence axioms

  t3. $p +\!\!+\, \emptyset = \emptyset +\!\!+\, p = p$

  t4. $(p +\!\!+\, q) +\!\!+\, r = p +\!\!+\, (q +\!\!+\, r)$

  t5. $x +\!\!+\, x = x$

- Infinite traces

  t6. $x +\!\!+\, x +\!\!+\, x +\!\!+ \ldots = x +\!\!+\, \infty$ (for an infinite sequence of x's)

  t7. $\infty +\!\!+\, p = \infty$

  t8. $p = q \Leftrightarrow \langle \forall r\ :\ r \text{ is finite }\ :\ r \text{ prefix of } p \Leftrightarrow r \text{ prefix of } q \rangle$
  where r prefix of $p \equiv \langle \exists p'\ ::\ p = r +\!\!+\, p' \rangle$.

We briefly discuss the definition.

Axioms t1 and t2 describe how traces are constructed. There are three types of 'primitive' traces: an empty trace, '$\emptyset$'; a special infinite trace, '∞'; and traces consisting of a single element. (It is assumed that $\emptyset$ and ∞ are not in X.) More complex traces can be constructed through the '$+\!\!+$' operator, called concatenation.

According to axiom t3, $\emptyset$ is the (left and right) neutral element of concatenation. Furthermore, concatenation is associative according to axiom t4.

As mentioned before, we do not make timing assumptions. Therefore, consecutive observations of the same state do not contain any more information than a single observation of that state. Hence, we may just as well collapse consecutive identical elements into a single instance of that element; this is expressed by axiom t5. The presence of consecutive identical states is called *stuttering*, and we refer to this axiom as the *stuttering axiom*.

We do not want to collapse infinite sequences (generated by non-terminating computations) into single elements; therefore, we introduce a special symbol '∞' and axiom t6. This axiom can be considered a technicality.

Finally, although we can observe an execution for an arbitrarily long time period, we cannot spend 'infinite time.' Therefore, if a computation is already infinite, we do not care what happens afterwards. This is captured by axiom t8, which says that two traces are equal if all their finite prefixes are equal. Hence, if p is an infinite trace, then $p +\!\!+ q$ is equal to $p +\!\!+ r$. In this context, an infinite trace is one that requires infinitely many construction steps. '∞' represents an infinite trace, but requires only a single construction step; therefore, it has its own variant of this axiom, t7.

Here is a normal form for finite traces.

Let *Trace* = **trace of** (X). A trace representation is in normal form if it has one of the following forms, for arbitrary $x, y \in X$ and $p \in$ *Trace*. $(\emptyset, \infty \notin X)$

`t1.` $\emptyset$, ∞, x.

`t2.` $x +\!\!+ y$, where $x \neq y$.

`t3.` $x +\!\!+ (y +\!\!+ p)$, where $x \neq y$ and $y +\!\!+ p$ is in normal form.

`t4.` $x +\!\!+ \infty$.

Hence, in the normal form, traces are written in right-associative form, $x +\!\!+ p$; repeating elements are removed; and empty traces are removed as much as possible. This normal form follows the standard procedure: The 'primitive' traces constructed by t1 (note the typographical distinction between t1 and `t1`) are always in normal form; for other traces (those constructed by t2), the normal form is obtained by favoring the right-hand side of t3–t7, until none of these axioms can be applied. It is straightforward to check that none of the left-hand sides of t3–t7 match traces of the form `t1`–`t4`. That every finite trace can indeed be written in a unique normal form requires a somewhat tedious, but not difficult, proof; we do not give this proof, because it is not important for our purposes, but an example of a similar proof is given in Appendix A.

We will not give a formal description of a model for the ADT, but only an informal description to make it plausible that such a model exists. Model a trace by a list of elements, but assume that the elements can only be observed by 'walking through' the list from the beginning. Then it is straightforward to only report an element when it differs from its predecessor, as in axiom t5. Also, that way, traces can only be distinguished if they differ in a finite prefix, as in t8. Finally, reporting only new elements does not work if there is an infinite sequence of identical elements; if such a sequence can be detected, we can report the ∞ trace. However, depending on how traces are specified, it is not necessarily possible to determine whether a trace can be written with ∞ or not.

The above ADT defines a type **trace of** (X) for arbitrary type X. We have already said that we wanted to represent computations by traces of states, hence the following definition.

DEFINITION 2.4 *(Computation)*

The type of computations is *Trace* = **trace of** $(State)$.

□

(*State* is the set of all states, including $\bot$.) Unless otherwise stated, the term trace refers to an element of *Trace*.

EXAMPLE 2.5

Although we have not yet formally specified what the relation between programs and traces is, we can give an informal example. Consider program

$$u\uparrow; v\uparrow; u\uparrow; v\downarrow; u\downarrow$$

Recall that an assignment like $u\uparrow$ is a state transformer that takes a state σ and changes it to $\sigma[u \mapsto \mathbf{true}]$. If the initial state of the execution is σ, this program generates the following trace (we have written $\sigma[u]$ for $\sigma[u \mapsto \mathbf{true}]$ and $\sigma[\neg u]$ for $\sigma[u \mapsto \mathbf{false}]$):

$$\sigma[u] +\!\!+ \sigma[u][v] +\!\!+ \sigma[u][v][u] +\!\!+ \sigma[u][v][u][\neg v] +\!\!+ \sigma[u][v][u][\neg v][\neg u]$$

Observing that $\sigma[u][\neg u] = \sigma[\neg u]$ etc., this is equivalent to

$$\sigma[u] +\!\!+ \sigma[u][v] +\!\!+ \sigma[u][v] +\!\!+ \sigma[u][\neg v] +\!\!+ \sigma[\neg u][\neg v]$$

Since the second and third states in this trace are identical, we can remove one of them using the stuttering axiom, resulting in

$$\sigma[u] +\!\!+ \sigma[u][v] +\!\!+ \sigma[u][\neg v] +\!\!+ \sigma[\neg u][\neg v]$$

Hence, according to the definition of traces, the second $u\uparrow$ does not contribute to the computation: it might as well be removed. This is indeed what we would intuitively expect from this program. If $\sigma \neq \sigma[v]$, the last representation of the trace is in normal form.

□

We extend projection to traces, by applying the projection to each state in the trace. (I.e., $(\sigma +\!\!+ p)\lfloor A = (\sigma\lfloor A) +\!\!+ (p\lfloor A)$, etc.) The projection of $\bot$ onto any set is just $\bot$, so that projection can never remove $\bot$ from a trace. Furthermore, projection is extended to sets of traces by applying the projection to each element trace.

EXAMPLE 2.6

Take the trace from the previous example. Assume that $\sigma(v) = \mathbf{false}$. Then, if we project on $\{u, v\}$, we get

$$[u][\neg v] +\!\!+ [u][v] +\!\!+ [u][\neg v] +\!\!+ [\neg u][\neg v]$$

where $\tau = [u]$ stands for the function with domain $\mathcal{D}(\tau) = \{u\}$ and $\tau(u) = \mathbf{true}$.

If we project on just $\{u\}$, we get

$$[u] +\!\!+ [u] +\!\!+ [u] +\!\!+ [\neg u]$$

which, by t5, is

$$[u] +\!\!+ [\neg u]$$

□

2.3 Programs

Normally, the syntax and type rules of a programming language define which objects are programs. However, in this section we define a special *Program* data type, independently of a specific programming language. A programming language can then be seen as a special notation for objects of *Program* type. Using this data type has the obvious advantage that we can talk about programs without reference to the language they are written in; in particular, it allows us to compare programs written in different languages. The *Program* data type is also convenient when we define environments. Programs can be executed, and then produce computations (traces); execution of programs is the subject of Chapter 3.

Although the *Program* data type is independent of a specific programming language, we have chosen a data type that can support the constructs that are of interest to us in this text; there may very well be programming languages that do not easily map to the *Program* data type. Since we have already introduced a data type for the data aspects of programs (namely, *State*), we now concentrate on the control aspects of programs. In particular, we want our data type to support the following control aspects.

- sequential composition;
- choices;
- non-determinism;
- parallel composition; and
- rendezvous synchronization.

Below we define an ADT '**tree of** (Nt, Et),' which supports these operations. Here, Nt and Et are arbitrary types on which the ADT depends; later we make specific choices for Nt and Et to represent programs. (Why we call the ADT a tree is explained below.) Actually, the ADT depends on another arbitrary type called *Label*. But since the only important property of *Label* is that its elements can be distinguished from one another, we will mostly ignore it. The ADT uses some sets and bags; a bag (also called a multiset) is similar to a set, but can contain more than one copy of an element. As mentioned in Section 1.3, to simplify notation, we sometimes omit the braces around elements of a set or a bag. E.g., we write $\mathsf{ext}(\emptyset; t)$ instead of $\mathsf{ext}(\emptyset; \{t\})$, and $T \cup t$ or T, t instead of $T \cup \{t\}$.

As with the ADT for traces, we point out that the purpose of the ADT is to enable us to refer unambiguously to trees; we have no intention of using the axioms to give formal proofs for lots of properties of trees. In fact, we will almost never compare trees directly, but instead 'execute' them and compare the resulting computations.

Let *Tree* = **tree of**(Nt, Et). The following axioms hold for any $f, g \in Nt$; $e \in Et$; $t, t_0, t_1, \ldots \in$ *Tree*; $S, T \in$ **set of**(*Tree*); $U, V \in$ **bag of**(*Tree*); and $\ell \in$ *Label.*

- Constants and constructors

  T1. $\emptyset \in \mathit{Tree}$
  $f \in \mathit{Tree}$
  $\mathsf{sync}_\ell(f; g) \in \mathit{Tree}$

  T2. $\mathsf{ext}(t; T) \in \mathit{Tree}$, if $T \neq \{\}$.

  T3. $e \rightarrow t \in \mathit{Tree}$

  T4. $\mathsf{par}(U) \in \mathit{Tree}$, if $U \neq \{\}$.

- Equivalence axioms

  T5. $\mathsf{ext}(\mathsf{ext}(t_0; T); S) = \mathsf{ext}(t_0; \langle \bigcup t \; : \; t \in T \; : \; \{\mathsf{ext}(t; S)\}\rangle)$
  or, in its simplified form:
  $\mathsf{ext}(\mathsf{ext}(t_0; t_1, \ldots, t_k); S) = \mathsf{ext}(t_0; \mathsf{ext}(t_1; S), \ldots, \mathsf{ext}(t_k; S))$

  T6. $\mathsf{ext}(e \rightarrow t; S) = e \rightarrow \mathsf{ext}(t; S)$

  T7. $\mathsf{ext}(\emptyset; t) = t$

  T8. $\mathsf{ext}(t; T \cup \mathsf{ext}(\emptyset; S)) = \mathsf{ext}(t; T \cup S)$

  T9. $\mathsf{par}(U \cup \mathsf{par}(V)) = \mathsf{par}(U \cup V)$

  T10. $\mathsf{par}(t) = t$

First we explain the ADT without reference to programs. The ADT is easiest explained in terms of a model for it. As the name suggests, we choose trees (with some special properties) as model. Since trees are often more easily understood with a picture, we present graphical descriptions of some of the trees we use. The pictures are for purpose of illustration only: the actual description of trees is with the constructors T1–T4.

Any tree t can be drawn as simply: .

The trees are labeled: node labels have type Nt, edge labels have type Et. (Some nodes have an additional identifying label of type *Label.*)

Axiom T1 defines the nodes of the tree.

- '$\emptyset$' is a special empty node (or empty tree). As before, we assume that $\emptyset \notin Nt$. The graphical representation is

  ○

- A node label by itself forms a node. We show it as

  $f\,\bullet$

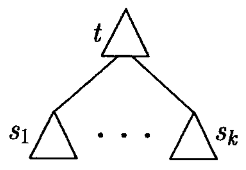
- The third, and last, type of node is called a ‘sync’ node. This type of node, written as $\mathsf{sync}_\ell(f;g)$, has two node labels (f and g) and an extra identifying label (ℓ). It is drawn as

$$f;g\times\ell$$

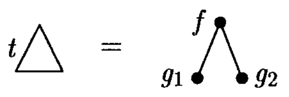
According to T1, a single node is considered a tree. We point out that in $\mathsf{sync}_\ell(f;g)$ the parentheses and the semi-colon are part of the notation: the node depends on three elements, $\ell \in \mathit{Label}$ and $f, g \in \mathit{Nt}$. In particular, there is no special meaning attached to the use of a semicolon — any other separator could have been used.

The ext constructor of T2 extends the tree, i.e., it adds any number of children to the tree. Graphically this is shown as

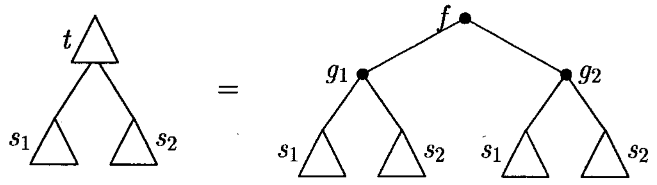
$\mathsf{ext}(t; s_1, \ldots, s_k) \quad =$



This differs from more ‘standard’ definitions of trees, where children can only be added to nodes, not to whole trees. Axiom T5 specifies what it means to add children (which are trees) to a tree: duplicates of the children are added to each leaf of the tree. For instance, if



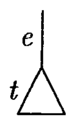
then



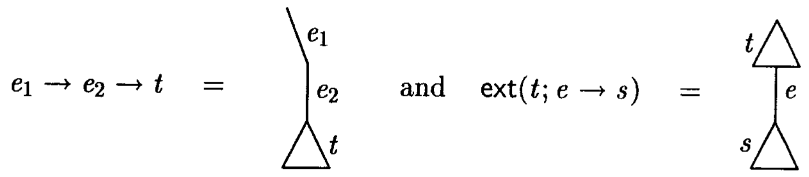
with two instances each of s_1 and s_2. In other words, T5 describes the associativity of ext. Recall that $\mathsf{ext}(t; s_1, s_2)$ is short for $\mathsf{ext}(t; \{s_1, s_2\})$. It is significant that the second argument of ext is a set: it means that we do not distinguish between identical children (i.e., all children are different), and that children are unordered. As with sync nodes, the parentheses and semicolon in $\mathsf{ext}(t; S)$ are part of the notation.

The ‘$\rightarrow$’ constructor of T3 adds an edge label to the root of a tree.

$e \rightarrow t \quad =$

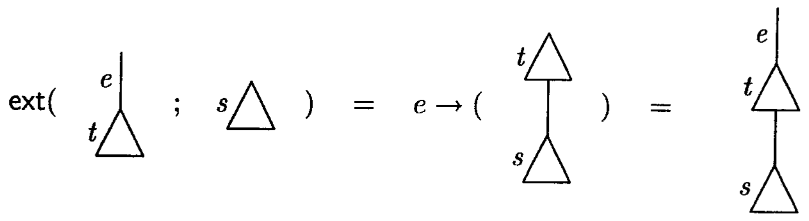


A tree with an edge label is itself a tree (to which another edge label can be added). Because of the types of its arguments, '$\rightarrow$' must be right-associative: $e_1 \rightarrow e_2 \rightarrow t = e_1 \rightarrow (e_2 \rightarrow t)$. An edge label can be considered a 'dangling' edge. Hence

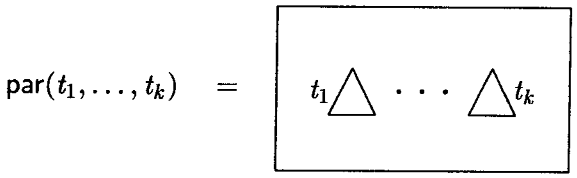


$e_1 \rightarrow e_2 \rightarrow t$
=
e_1
e_2
t
and
ext(t; $e \rightarrow s$)
=
t
e
s

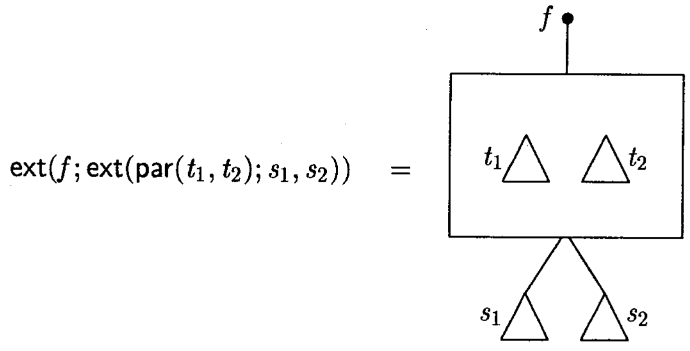
According to axiom T6, it does not matter whether edge labels or children are added first:



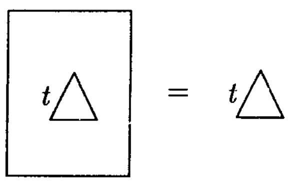
The par constructor of T4 allows us to consider a bag of trees as a single tree. In the graphical representation we draw a box around the bag of trees:



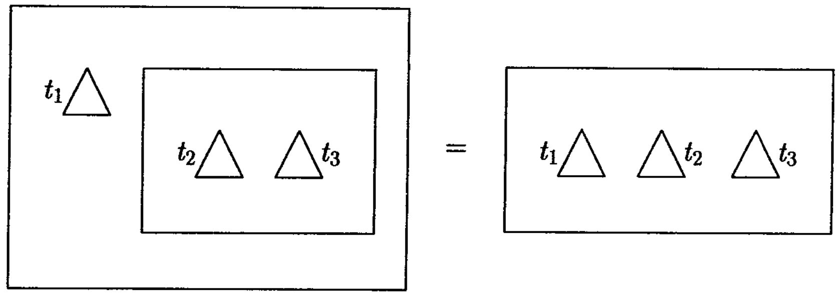
Since this is a single tree, it can be used together with other constructors; for instance



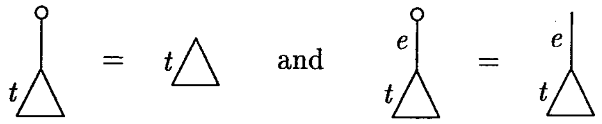
Again, the elements of par are unordered, but they need not all be distinct. We do not distinguish between a bag of a single tree and the tree by itself, as specified by axiom T10.



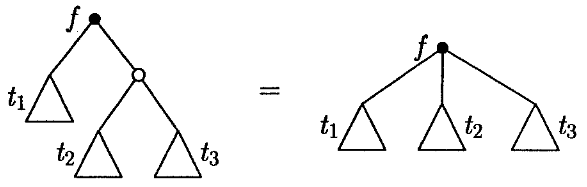
Axiom T9 specifies associativity of the par constructor. For instance



Finally, T7 and T8 say that empty nodes don't count: they can be inserted or removed anywhere in a tree. Basically, we use empty nodes as place holders if we do not yet know which normal node will be used. Examples of T7:



An example of T8:



In contrast with our definition of traces, we have no special axioms for infinite trees, although we will use some infinite trees (namely, trees with infinitely many extensions). In particular, we have no equivalent of t8 saying that only finite 'prefixes' of trees can be observed. The reason is that we do not observe trees directly at all: instead, in the next chapter we describe a procedure that associates certain traces with trees; these traces are then observed. The result is that only finite prefixes of trees can be observed, since they correspond to finite prefixes of traces. Hence, there is no need to complicate the definition of trees with axioms that specify this restriction explicitly.

Here is a normal form for trees.

Let *Tree* = **tree of**(Nt, Et); $f, g \in Nt$; $e \in Et$; $t \in$ *Tree*; $S \in$ **set of**(*Tree*); $U, V \in$ **bag of**(*Tree*); and $\ell \in$ *Label*.

- A tree representation is in normal form if it has one of the following forms.

  T1. $\emptyset$, f, $\mathsf{sync}_\ell(f; g)$.

  T2. $\mathsf{ext}(\emptyset; S)$, where S has more than one element and $t \in S$ is in internal normal form.

  T3. $\mathsf{ext}(n; S)$, where $t \in S$ is in internal normal form and n is f or $\mathsf{sync}_\ell(f; g)$.

  T4. $\mathsf{ext}(\mathsf{par}(U); S)$, where $t \in S$ is in internal normal form and $\mathsf{par}(U)$ is in normal form.

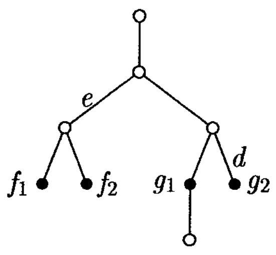
  T5. $e \rightarrow t$, where t is in normal form.

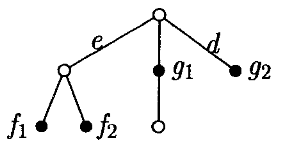
  T6. $\mathsf{par}(U)$, where U has more than one element and $t \in U$ is in normal form and $t \neq \mathsf{par}(V)$.

- A tree representation is in internal normal form if it is in normal form and not of the form described by T2.

Hence, in the normal form, extension is written in right-associative form: $\mathsf{ext}(n; S)$ with n a node, or $\mathsf{ext}(\mathsf{par}(U); S)$, but never $\mathsf{ext}(t; S)$ for other forms of t; nested par constructs are removed; and empty nodes are removed as much as possible. In what is called internal normal form, no trees of the form $\mathsf{ext}(\emptyset; \ldots)$ are allowed. Therefore, if $\mathsf{ext}(n; S)$ is in normal form, no $t \in S$ can start with a $\emptyset$ node. Hence, $\emptyset$ nodes can only occur as the root of a tree, following a labeled edge, or as the root of a tree in a par construct; in each of these cases, the $\emptyset$ node must have more than one child. In addition, $\emptyset$ nodes can occur as leaves, i.e., without any children at all.

EXAMPLE 2.7 *($\emptyset$ Nodes)*

The normal form of 

is 

□

As before, the normal form is constructed by application of two rules: Trees of a form described by T1 are in normal form; for all other trees (those constructed by T2–T4), the normal form is obtained by favoring the right-hand side of T5–T10. Once again, the proof that this is a normal form is omitted, because it is lengthy but straightforward, and not very important for our purposes.

The above ADT defines a type $\mathbf{tree\,of}(Nt, Et)$ for arbitrary types Nt and Et. In order to represent programs with trees, we make the following choices for Nt and Et.

DEFINITION 2.8 *(Program)*

The type of programs is $\mathit{Program} = \mathbf{tree\,of}(\mathit{State} \rightarrow \mathit{State}, \mathit{State} \rightarrow \mathit{Boolean})$.

□

We refer to elements of *Program* as programs or program trees, or, occasionally, if the context is clear, just trees (however, in the next chapter we will use trees with different choices for Nt and Et). $\mathit{State} \rightarrow \mathit{State}$ is the type of state transformers (mappings from states to states). As we have already seen, assignments can be considered state transformers. Because state transformers are used as nodes in program trees, we sometimes refer to them simply as nodes. $\mathit{State} \rightarrow \mathit{Boolean}$ is the type of guards. Hence, programs are trees with state transformers as node labels and guards as edge labels.

In the remainder of this section we define some terms that are useful when dealing with trees.

DEFINITION 2.9

- A tree of the form t or $\mathsf{ext}(t; \ldots)$ is called a tree *starting with* t; in particular, f and $\mathsf{ext}(f; \ldots)$ are trees starting with f.
- A tree of the form $e \rightarrow t$ is called a *guarded tree*.

□

The following definitions characterize what part of a state is important with respect to a guard or state transformer.

DEFINITION 2.10 *(changes)*

- If f is a state transformer, $\mathit{changes}(f)$ is defined as the set of variables u for which

  $u \in \mathit{changes}(f) \equiv$
  $\langle \exists \sigma \,:\, \sigma \in \mathit{State} \,:\, (u \notin \mathcal{D}(\sigma) \wedge u \in \mathcal{D}(f(\sigma))) \vee \sigma(u) \neq (f(\sigma))(u) \rangle$
- If f and g are state transformers, $\mathit{changes}(\mathsf{sync}_\ell(f; g))$ is defined as

$$\mathit{changes}(\mathsf{sync}_\ell(f; g)) = \mathit{changes}(f) \cup \mathit{changes}(g)$$

□

$\mathit{changes}(f)$ is the set of variables that f assigns to and changes. This includes variables that are assigned a new value, as well as variables that previously were not part of the state.

DEFINITION 2.11 *(depends)*

- If e is a guard, $\mathit{depends}(e)$ is defined as the set of variables u for which

$$u \in \mathit{depends}(e) \equiv \langle \exists \sigma, a \,:\, \sigma \in \mathit{State} \wedge a \in \mathbf{type}(u) \,:\, e(\sigma) \neq e(\sigma[u \mapsto a]) \rangle$$

- If f is a state transformer, *depends*(f) is defined as the set of variables u for which

$$\begin{aligned} u \in \mathit{depends}(f) \equiv \langle \exists \sigma, a \,:\, & \sigma \in \mathit{State} \land a \in \mathbf{type}(u) \,: \\ & (\sigma \in \mathcal{D}(f) \land \sigma[u \mapsto a] \notin \mathcal{D}(f)) \lor \\ & (f(\sigma)\lfloor \mathit{changes}(f) \neq f(\sigma[u \mapsto a])\lfloor \mathit{changes}(f) \\ \rangle & \end{aligned}$$

- If f and g are state transformers, *depends*$(\mathsf{sync}_\ell(f;g))$ is defined as

$$\mathit{depends}(\mathsf{sync}_\ell(f;g)) = \{\ell\} \cup \mathit{depends}(f) \cup \mathit{depends}(g)$$

□

depends(f) is the set of variables on which the assignment performed by f depends; its definition is clarified by the example below. $\ell \in \mathit{Label}$ is not a variable but nevertheless part of *depends*$(\mathsf{sync}_\ell(f;g))$, because, as we will see later, the execution of sync nodes depends on these labels.

EXAMPLE 2.12

- f is $u := v + w$: $\mathit{changes}(f) = \{u\}$ and $\mathit{depends}(f) = \{v, w\}$.
- f is $u := u$: $\mathit{changes}(f) = \{\}$ and $\mathit{depends}(f) = \{\}$ (note that, by convention, $u \in \mathcal{D}(\sigma)$ if $\sigma \in \mathcal{D}(f)$, because f refers to $\sigma(u)$).
- f is $u := u + 1$: $\mathit{changes}(f) = \{u\}$ and $\mathit{depends}(f) = \{u\}$.
- f is $u := c$ where c is a constant: $\mathit{changes}(f) = \{u\}$ and $\mathit{depends}(f) = \{\}$.
- f is $u := \frac{1}{v-w}$ where v and w can only be 0 or 1: $\mathit{changes}(f) = \{u\}$ and $\mathit{depends}(f) = \{v, w\}$. Note that if $\sigma \in \mathcal{D}(f)$, then $\sigma(v) \neq \sigma(w)$ (to avoid division by 0). Hence, changing a single variable, as in the definition of *depends*(f), results in $\sigma' \notin \mathcal{D}(f)$. This is the reason for the first disjunct in the definition of *depends*(f).

□

We extend *depends* and *changes* to program trees by taking the union of the corresponding sets for each guard and each node. This is easily formalizable with rules like

$$\mathit{depends}(\mathsf{ext}(f; t_1, \ldots, t_k)) = \mathit{depends}(f) \cup \mathit{depends}(t_1) \cup \ldots \cup \mathit{depends}(t_k)$$

We omit the remainder of that formalization.

DEFINITION 2.13 *(var)*

For a program t the set of its variables is defined as

$$\mathit{var}(t) = \mathit{depends}(t) \cup \mathit{changes}(t)$$

□

As we pointed out at the beginning of this section, a programming language can be seen as a special notation for objects of type *Program*. Hence, now that we have defined what type *Program* is, we should define how programming language constructs correspond to program trees. This is the subject of the next section.

2.4 Standard Programming Constructs

In the previous section we defined that programs are trees with state transformers as node labels and guards as edge labels:

$$\mathit{Program} = \textbf{tree of}\,(\mathit{State} \longrightarrow \mathit{State}, \mathit{State} \longrightarrow \mathit{Boolean})$$

In this section we describe how standard programming constructs, like assignments and sequential composition, correspond to such program trees. Since programs are trees, and because this is an operational semantics, there must be some way to 'execute' trees. The formal definition of program execution is the topic of Chapter 3; for the time being, we use an informal notion of execution to motivate the correspondence between programming constructs and trees. For the description below, recall that we write $u := a$ for the state transformer that maps σ to $\sigma[u \mapsto \sigma(a)]$. Likewise, if e is a boolean expression, we write e for the guard that maps σ to $\sigma(e)$. Below, let t_1 and t_2 be programs.

Informally, the idea of program execution is that a computation corresponds to a path through the tree, starting at the root and moving in the direction of leaves. We start the path with the initial state of the computation. Whenever we encounter a state transformer, we apply it to the state to get a new state. Hence, the simplest statements, such as **skip** and assignments, correspond to nodes.

- A **skip** is a node with the identity function as label:

  $$\textbf{skip} \quad = \quad x \mapsto x \qquad\qquad \bullet\,\textbf{skip}$$

  By analogy with the notation for assignments, we write **skip** to denote the identity state transformer.

- An assignment maps to a single node with the appropriate state transformer as label:

  $$u := a \quad = \quad u := a \qquad\qquad \bullet\, u := a$$

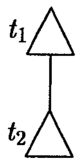
  (The first assignment is the notation used in programs, the second and third denote state transformers.)

Empty nodes, $\emptyset$, should have no effect on the computation whatsoever, because we want to use them merely as place holders. Although a **skip** does not modify the state, we have equated it with a state transformer rather than with $\emptyset$. The reason is that we do not want to remove statements from the tree, like we can often do with $\emptyset$ nodes; if a statement has no effect on the computation, then that should follow from the result of its execution, not from its a priori removal.

Once the state has been modified by a node, the execution moves to a child of the node. If there is more than one child, there is more than one way to continue the computation: the execution has reached a choice. Since there is no ordering among the children of a tree, the choice is non-deterministic.

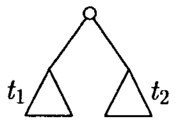
- Since execution of a tree is followed by execution of a child, sequential composition corresponds to the ordering between parent and child:

$$t_1; t_2 \quad = \quad \mathsf{ext}(t_1; t_2)$$



- The ext constructor can also be used to denote a non-deterministic choice between t_1 and t_2:

$$t_1 \,\Box\, t_2 \quad = \quad \mathsf{ext}(\emptyset; t_1, t_2)$$



However, this non-deterministic branch is not a statement that we actually use in our notation.

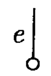
From the definition of sequential composition it follows that sequential composition is associative:

$$\begin{aligned}
& (t_1; t_2); t_3 \\
= \quad & \{\text{definition}\} \\
& \mathsf{ext}(\mathsf{ext}(t_1; t_2); t_3) \\
= \quad & \{\text{T5}\} \\
& \mathsf{ext}(t_1; \mathsf{ext}(t_2; t_3)) \\
= \quad & \{\text{definition}\} \\
& t_1; (t_2; t_3)
\end{aligned}$$

Whenever the execution encounters a guard, the guard is evaluated, i.e., applied to the current state. Only if the guard yields **true** can the computation continue beyond the guard.

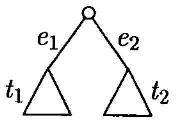
- A wait [e] is a statement that cannot be passed unless e holds:

$$[e] \quad = \quad e \rightarrow \emptyset$$



- By combining guarded trees with non-deterministic branching, we obtain the standard selection statement with guarded commands:

$$\begin{array}{lll} [\ e_1 \longrightarrow & t_1 & \\ \,\Box\ e_2 \longrightarrow & t_2 & = \quad \mathsf{ext}(\emptyset; e_1 \rightarrow t_1, e_2 \rightarrow t_2) \\ ] & & \end{array}$$



Note that the intention is that the guards are evaluated before the choice between the children is made; after guard evaluation, we can only choose between guards that yielded **true**.

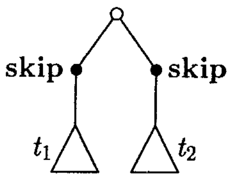
It is possible that the execution reaches a point where all continuations start with a **false** guard; in that case the computation ends with deadlock ($\perp$), as explained in Section 2.2.

If a choice is made non-deterministically, potentially the worst possible choice can be made. Therefore, this type of non-determinism is sometimes referred to as *demonic* non-determinism. The opposite of demonic non-determinism is *angelic* non-determinism, where we can count on the best possible choice being made. Angelic non-determinism cannot easily be implemented, so that it is unlikely to be found in an operational semantics. However, from our definition of the selection statement it follows that the non-determinism provided by the choice between multiple children is not quite demonic either: a child starting with a false guard cannot be selected. Even without a definition of what constitutes a good or a bad choice, a choice for a false guard would certainly be rather bad, as it would lead to deadlock immediately. Hence, our form of non-determinism is more 'benevolent' than demonic non-determinism; we call it *guarded* non-determinism.

Above, we gave an example definition of a non-deterministic branch, $t_1 \Box t_2$. If '$\Box$' is supposed to express a demonic choice, the given definition, $\mathsf{ext}(\emptyset; t_1, t_2)$, is not correct, because guarded non-determinism is used to make the choice. We can force a demonic choice by making sure that neither of the alternatives is guarded by a **false** guard. Here are two ways to achieve that. (If both alternatives are guarded by a **false** guard the choice is also demonic, because either choice is equally bad. However, we cannot enforce this situation without changing the meaning of the program.)

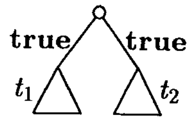
- Assuming that the presence of a **skip** does not alter the meaning of a program, the following defines a demonic choice between t_1 and t_2:

$$t_1 \Box t_2 \quad = \quad \mathsf{ext}(\emptyset; \mathsf{ext}(\mathbf{skip}; t_1), \mathsf{ext}(\mathbf{skip}; t_2))$$



- The same effect is obtained by insertion of extra **true** guards:

$$t_1 \,\square\, t_2 \quad = \quad \mathsf{ext}(\emptyset; \mathbf{true} \rightarrow t_1, \mathbf{true} \rightarrow t_2)$$



For convenience when discussing program trees, we define the following terms.

DEFINITION 2.14

- A tree of the form $\mathsf{ext}(f; t_1, \ldots, t_k)$ is called a *choice* if $k > 1$. $t_1, \ldots, t_k$ are called *alternatives*.
- A *guarded choice* is a choice where each alternative is a guarded tree or a tree starting with a sync node; an *unguarded choice* is a choice where at least one alternative is not guarded and does not start with a sync node.
- A guarded tree that is not part of a choice (e.g., $\mathsf{ext}(f; e \rightarrow t)$ is not a choice) is called a *wait*.

□

We often write non-terminating programs. Since a computation corresponds to a path through the tree, a non-terminating computation must correspond to a non-terminating path, which requires a non-terminating sequence of extensions (ext constructors). In other words, a non-terminating loop is represented by its infinite unrolling. As mentioned before, only finite prefixes of a trace, i.e., of a computation, can be observed. Therefore, since trees are observed only by observing their computations, only finite 'prefixes' of infinite trees are observable. This is what allows us to treat infinite trees somewhat informally (see also Appendix A).

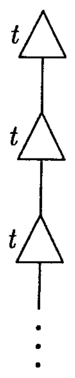
- Infinite loops correspond to infinite trees:

$$*[t] \quad = \quad \mathsf{ext}(t; \mathsf{ext}(t; \mathsf{ext}(t; \ldots)) \ldots)$$

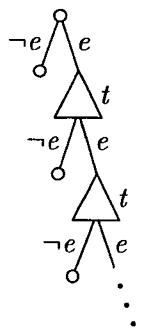
- A guarded loop can terminate after any number of iterations, or never terminate. Therefore, a guarded loop is also an infinite tree, but there are leaves at finite depth:

$$*[e \longrightarrow t] \quad = \quad \mathsf{ext}(\emptyset; \neg e \rightarrow \emptyset, e \rightarrow \mathsf{ext}(t; \neg e \rightarrow \emptyset, e \rightarrow \mathsf{ext}(t; \ldots)) \ldots)$$

$*[t] \quad =$



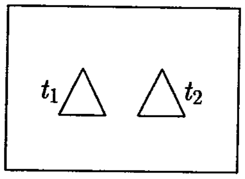
$*[e \longrightarrow t] \quad =$



Considering the name we gave it, it should hardly come as a surprise that par constructs are used to express parallel composition.

- Parallel composition:

$$t_1 \mathbin{//} t_2 \quad = \quad \mathsf{par}(t_1, t_2)$$



We have chosen to model parallel execution with non-deterministic interleaving, meaning that we non-deterministically choose between the trees, execute one 'step' of that tree, and repeat the process. The details are not important here, and are explained in the next chapter.

From the associativity of par, as expressed by axiom T9, it follows immediately that parallel composition is associative. Since par takes a bag as argument, the elements of which are not ordered, it also follows that parallel composition is commutative.

The execution of sync nodes differs from the execution of regular nodes: sync nodes can only be executed in pairs, formed by sync actions with identical ℓ labels (as in $\mathsf{sync}_\ell(f; g)$). With a pair of sync nodes, we associate a state transformer called the *match*. The execution of a pair of sync nodes is then the execution of the corresponding match.

Definition 2.15 *(Matching* sync *Nodes)*

The match of $\mathsf{sync}_\ell(f_1; f_2)$ and $\mathsf{sync}_\ell(g_1; g_2)$ is the state transformer

$$f_2 \circ g_2 \circ f_1 \circ g_1$$

□

The result of the match depends on which sync node is taken first. If, however, $f_2 \circ g_2 = g_2 \circ f_2$ and $f_1 \circ g_1 = g_1 \circ f_1$, then the match is independent of the ordering of the sync nodes; in that case we say that the match is *commutative.*

In our programming notation we use *rendezvous synchronization* (also called *synchronous* or *slack-less* synchronization), where a synchronization action can only complete when executed simultaneously with another (matched) synchronization action. If synchronization actions end with a sync node, they indeed exhibit this rendezvous behavior, because sync nodes are executed as a pair in a single step. It is possible to combine synchronization with exchange of data; in that case, the synchronization actions are called *communication actions.* The two state transformer labels of sync nodes can be used to represent such an exchange of data.

The programming notation must identify which synchronization actions can be matched; often an object called a *channel* is used for this. However, the exact mechanism for this in the programming language is not important for our purposes; therefore, in this text we usually identify pairs by using a label ℓ as subscript of each synchronization action, as in C_ℓ. This same label is then used for the corresponding sync nodes.

- Let C_ℓ be a synchronization action that can be matched with another, identical, C_ℓ action. Since there is no data involved, both state transformer labels are identity functions:

$$C_\ell \quad = \quad \mathsf{sync}_\ell(\mathbf{skip};\mathbf{skip}) \qquad\qquad \ell\boldsymbol{\times}\mathbf{skip};\mathbf{skip}$$

  Obviously, the match of two of these actions is commutative and equal to **skip**.

- Let $C_\ell!a$ be a communication action, called a *send*, that can be matched with a communication action $C_\ell?u$, called a *receive*. Here, u is a variable and a is an expression of the type of u. To represent these actions with trees we introduce an extra variable C_ℓ in the state:

$$C_\ell!a \quad = \quad \mathsf{sync}_\ell(C_\ell := a;\mathbf{skip}) \qquad\qquad \ell\boldsymbol{\times}C_\ell := a;\mathbf{skip}$$

$$C_\ell?u \quad = \quad \mathsf{sync}_\ell(\mathbf{skip};u := C_\ell) \qquad\qquad \ell\boldsymbol{\times}\mathbf{skip};u := C_\ell$$

  The match of these two actions is commutative (because each pair of functions involves an identity function) and equal to

$$(u := C_\ell \circ \mathbf{skip} \circ C_\ell := a \circ \mathbf{skip}) \quad = \quad (u := C_\ell \circ C_\ell := a)$$

  This has the same effect on u as $u := a$ (the effect on C_ℓ is irrelevant because C_ℓ is not used anywhere else). For this reason, a communication is sometimes called a *remote assignment.*

The send and receive are the standard CSP communication actions. They move data from the sender to the receiver. It is possible to define more complex exchanges of data, such as a *swap*. In a swap, data is moved in both directions; hence, it is somewhat like the combination of a send and a receive. We'll use here the notation $C!a?u$, but we could just as well have used $C?u!a$ or $C!?(a,u)$ or so — this denotes a new type of action, not a special case of the earlier send and receive actions.

- Let $C_\ell!a_1?u_1$ and $C_\ell!a_2?u_2$ denote swap actions that can be matched with one another. We introduce two extra variables in the state, C_ℓ^1 and C_ℓ^2.

$$C_\ell!a_1?u_1 \quad = \quad \mathsf{sync}_\ell(C_\ell^1 := a_1; u_1 := C_\ell^2) \qquad\qquad \ell\boldsymbol{\times}C_\ell^1 := a_1; u_1 := C_\ell^2$$

$$C_\ell!a_2?u_2 \quad = \quad \mathsf{sync}_\ell(C_\ell^2 := a_2; u_2 := C_\ell^1) \qquad\qquad \ell\boldsymbol{\times}C_\ell^2 := a_2; u_2 := C_\ell^1$$

The match is again commutative (because a_1 and a_2 do not depend on C_ℓ^1 and C_ℓ^2), and equal to

$$u_1 := C_\ell^2 \circ u_2 := C_\ell^1 \circ C_\ell^1 := a_1 \circ C_\ell^2 := a_2$$

The effect on u_1 and u_2 is that of the multiple assignment

$$u_1, u_2 := a_2, a_1$$

Let $C_\ell!$ and $C_\ell?$ be matching synchronization actions. If $C_\ell!$ is executed but no matching $C_\ell?$ is available, the $C_\ell!$ action cannot complete — it suspends. In a program it is sometimes desirable to avoid this situation. For this, an operator called the *probe* [12] can be used. The probe of $C_\ell!$, which we write as $\#C_\ell!$, is an expression that evaluates to **true** if it is guaranteed that $C_\ell!$ will complete rather than suspend; this guarantee can be given if and only if a $C_\ell?$ action has already been reached. The meaning of $\#C_\ell?$ is symmetric: it is true if a $C_\ell!$ action has been reached. (Note: more common notations for the probe are $\overline{C_\ell}$ and $\#C_\ell$, where the context makes clear whether it is the sending (!) or the receiving (?) side of the channel that is being probed.)

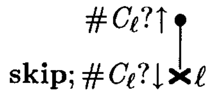
The definitions of synchronization and communication actions given above do not allow for probing. However, they are easily modified to make a probe possible.

- Let $C_\ell!$ be a synchronization action that can be matched with $C_\ell?$. ('!' and '?' are used so that we can distinguish the two probes, even though no data is exchanged.) To represent probes in a program tree, we consider '$\#C_\ell!$' and '$\#C_\ell?$' variables (with admittedly unusual names) rather than expressions. The following definitions of $C_\ell!$ and $C_\ell?$ give these variables the values corresponding to the probe:

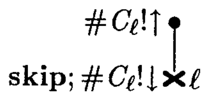
$$C_\ell! \quad = \quad \mathsf{ext}(\#C_\ell?\uparrow; \mathsf{sync}_\ell(\mathbf{skip}; \#C_\ell?\downarrow))$$

$$C_\ell? \quad = \quad \mathsf{ext}(\#C_\ell!\uparrow; \mathsf{sync}_\ell(\mathbf{skip}; \#C_\ell!\downarrow))$$

$C_\ell!$ =



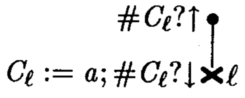
$C_\ell?$ =

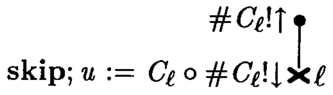


- Communication actions are extended to allow probing in exactly the same way:

$$C_\ell!a \quad = \quad \mathsf{ext}(\#C_\ell?\uparrow; \mathsf{sync}_\ell(C_\ell := a; \#C_\ell?\downarrow))$$

$$C_\ell?u \quad = \quad \mathsf{ext}(\#C_\ell!\uparrow; \mathsf{sync}_\ell(\mathbf{skip}; u := C_\ell \circ \#C_\ell!\downarrow))$$

$C_\ell!a \quad =$ 

$C_\ell?u \quad =$ 

These definitions still exhibit the desired rendezvous synchronization, because they end with a sync. In both cases, the match of the synchronization actions is still commutative. Note that, for instance, $\#C_\ell?$ is only used in $C_\ell!$, not in $C_\ell?$. Other parts of the program can read $\#C_\ell?$, but cannot assign to it. It follows that if $\#C_\ell?$ is not read anywhere, then it may just as well be omitted, meaning that the old definition (without probe) of $C_\ell!$ can be used. This is independent of whether the old or new definition of $C_\ell?$ is used. (That an unused variable can be omitted seems intuitively correct; we will formally prove it in Chapter 4.) In practice, our programs always have the property that at most one of $\#C_\ell?$ and $\#C_\ell!$ is used, never both. Therefore, we can always use at least one of the old definitions.

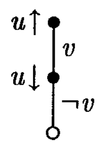
This completes the definition of standard programming constructs in terms of program trees, thus establishing conventional programming language constructs as alternative notations for trees. Indeed, we will often use programming notation to denote trees, just as we write assignments for state transformers.

EXAMPLE 2.16

$u\uparrow$; [v]; $u\downarrow$; [$\neg v$]

denotes the tree

ext($u\uparrow; v \rightarrow$ ext($u\downarrow; \neg v \rightarrow \emptyset$))



□

Note that all data aspects of a program are captured by state transformers and guards, whereas the structure of the tree captures the control aspects, thus providing the separation of data and control we mentioned at the beginning of this chapter. Together with the semantics of expressions (i.e., definitions like $\sigma(a_1 + a_2) = \sigma(a_1) + \sigma(a_2)$) given in Section 2.1, this section is the only part of the semantics that refers to a particular programming language. All other parts of this and the next chapter are concerned with language-independent objects such as traces and trees. Hence, to use our semantics to define the meaning of a particular programming language, one need only define the mappings from expressions to states and guards, from assignments to state transformers, and from program constructs to trees.

Chapter 3

Semantics: Execution and Refinement

In the previous chapter we defined programs and computations, but not the relationship between them. In this chapter we define program execution as a function that maps programs to sets of computations. We also define how programs can be composed with environments, before they are executed. Finally, the result of program execution is used to define an implementation relation between programs.

3.1 Program Execution

In the previous chapter we have equated programs with trees, and computations with traces. As mentioned before, when a program is executed, the result is one or more computations. In this section we define how a set of traces is generated from a program tree. We begin by defining a function *exec* of the following type:

$$exec : Program, State \rightarrow \textbf{tree of}\,(State, Boolean)$$

Hence, *exec* takes two arguments, a program and a state; the state is the initial state of the execution. Evaluation of $exec(t, x)$ yields a tree with states as node labels and truth values as edge labels; this tree is called an *execution tree*. The execution tree is not the computation itself, but an intermediate step; later we define the correspondence between execution trees and traces.

We now define for each form of tree (in normal form) how it is transformed into an execution tree. The reader may want to compare this with the informal description in Section 2.4, where program execution was described as following a path through the program tree, applying state transformers and evaluating guards along the way. *exec* does indeed do that, except that all paths are generated at once.

Below, let $x \in State$ be the initial state. Each item discusses a particular form of tree.

- $\emptyset$

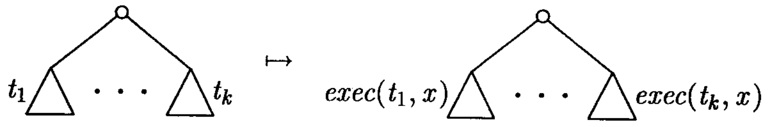
  Empty nodes should have no effect on the execution at all. Therefore, the result of execution is just an empty node again.

E1 $$exec(\emptyset, x) = \emptyset$$

$$\circ \mapsto \circ$$

- $\mathsf{ext}(\emptyset; t_1, \ldots, t_k)$
  If the empty node has children, the execution tree has corresponding children.

E2 $$exec(\mathsf{ext}(\emptyset; t_1, \ldots, t_k), x) = \mathsf{ext}(\emptyset; exec(t_1, x), \ldots, exec(t_k, x))$$



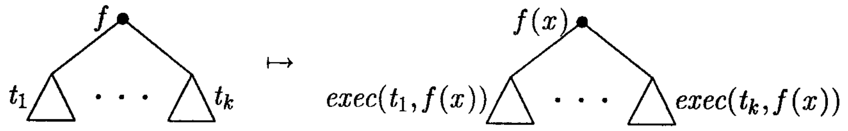
- f
  A state transformer, i.e., a node in the program tree, is applied to the current state (x) to yield a new state, which forms a node in the execution tree.

E3 $$exec(f, x) = f(x)$$

$$f\bullet \mapsto \bullet f(x)$$

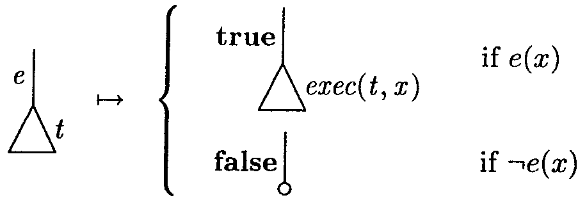
- $\mathsf{ext}(f; t_1, \ldots, t_k)$
  If the program continues after f, the execution continues with the modified state, $f(x)$.

E4 $$exec(\mathsf{ext}(f; t_1, \ldots, t_k), x) = \mathsf{ext}(f(x); exec(t_1, f(x)), \ldots, exec(t_k, f(x)))$$

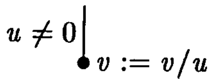
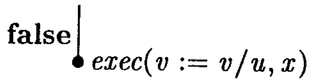


- $e \longrightarrow t$
  When a guard is encountered, it should be applied to the current state, yielding a boolean that forms an edge label in the execution tree. If $e(x)$ holds, the execution continues with t with the same state x; if $\neg e(x)$ holds, t cannot be reached.

E5 $$exec(e \longrightarrow t, x) = \begin{cases} \textbf{true} \longrightarrow exec(t, x) & \text{if } e(x) \\ \textbf{false} \longrightarrow \emptyset & \text{if } \neg e(x) \end{cases}$$



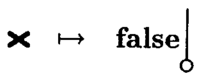
Note that we cannot just define $exec(e \to t, x) = e(x) \to exec(t, x)$, because, if $\neg e(x)$, then $exec(t, x)$ may contain function evaluations that should not be reached because they are undefined. For instance, if $x(u) = 0$,

 cannot be defined equal to 

because the last tree has no meaning.

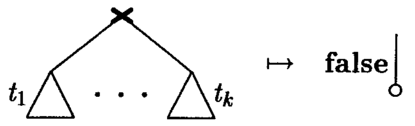
- $\mathsf{sync}_\ell(f; g)$
  Execution of sync nodes must occur in pairs. Hence, encountering only a single sync node is just like encountering a **false** guard. Let sync stand for $\mathsf{sync}_\ell(f; g)$.

E6 $\qquad exec(\mathsf{sync}, x) = \textbf{false} \to \emptyset$



- $\mathsf{ext}(\mathsf{sync}_\ell(f; g); t_1, \ldots, t_k)$
  As with **false** edges, whatever follows a single sync node should not be executed.

E6′ $\qquad exec(\mathsf{ext}(\mathsf{sync}; t_1, \ldots, t_k), x) = \textbf{false} \to \emptyset$

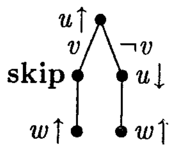


For both **false** guards and single sync nodes we have inserted a **false** edge in the execution tree, and omitted any children because they should not be reachable. When a trace is formed from the execution tree, it should not pass any edges labeled **false**. Therefore, if all edges leading out of a node are labeled **false**, the path cannot continue, a situation we have earlier called deadlock. When we describe how traces are formed from the execution tree, this will be formalized by adding a deadlock state, $\perp$, to the trace.

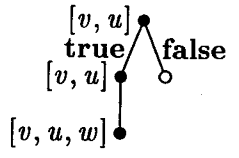
EXAMPLE 3.1

Consider program

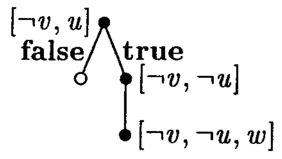
```
            u↑;
            [ v   ⟶  skip
t    =      [] ¬v  ⟶  u↓        =
            ];
            w↑
```



For brevity, write a state $x[v \mapsto \mathbf{true}][u \mapsto \mathbf{false}]$ as $[v, \neg u]$. Here are two execution trees, resulting from $exec(t, [v, \neg u])$ and $exec(t, [\neg v, \neg u])$, respectively. Assume $w \notin \mathcal{D}(x)$.



$exec(t, [v, \neg u])$



$exec(t, [\neg v, \neg u])$

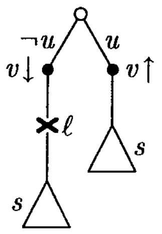
Note that these execution trees do *not* start with the initial state.

□

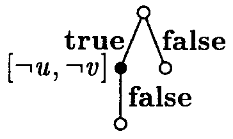
EXAMPLE 3.2

Consider the following program t, where C_ℓ is a synchronization action.

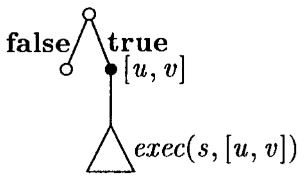
$$t \;=\; \begin{array}{l} [\ \neg u \longrightarrow v\!\downarrow;\ C_\ell \\ []\ u \longrightarrow v\!\uparrow \\ ]; \\ s \end{array} \;=$$



Execution with initial states $[\neg u]$ and $[u]$ results in

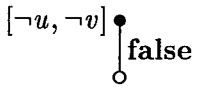


$exec(t, [\neg u])$



$exec(t, [u])$

The empty node at the root remains an empty node, thus having no effect on the execution. Note that the execution tree $exec(t, [\neg u])$ has a subtree



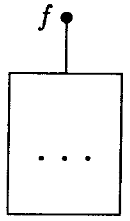
.

If a path reaches this subtree, it cannot continue because a path cannot pass a **false** edge and there are no other edges; hence, such a path would end with deadlock. The other **false** edge does not lead to deadlock, because there is another edge that is not labeled **false**.

□

There are still two forms of tree for which we have not defined the corresponding execution trees, both involving a par construct. There is no obvious direct way to follow a path through a par construct. Therefore, instead of executing the par, we execute a different tree called its *non-deterministic interleaving*. We cannot just replace the par by its interleaving before we execute the tree, because we do not want to interleave parts of the tree that cannot be reached because of **false** edges; hence, the interleaving depends on the state.

Informally, interleaving a bag $\{t_1, \ldots, t_k\}$ means that we choose one of the trees t_i, execute a single action of it, put the remaining part of t_i back in the bag, and repeat this process until the bag is empty. Here, an action corresponds to a single node or to a guard. We can make the choice of t_i with the normal non-deterministic choice provided by an ext construct. To take the first action of t_i and put the remainder back in the bag, we will use a function $\mathit{ileave}(t_i; U)$, where U is the original bag of programs without t_i. Hence, if t_i starts with a node f, the result of $\mathit{ileave}(t_i; U)$ typically has the form

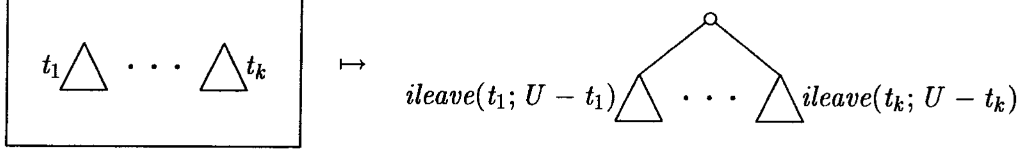


Note that $\mathit{ileave}(t_i; U)$ does not perform a complete interleaving: it only takes out the first action of t_i and creates a bag, i.e., a par, with the remaining trees (unless the new bag is empty). This is done because it isn't known whether the remaining trees are reachable until the first action of t_i has been executed. The definition of *ileave* is given in the next section; here we only need to know that $\mathit{ileave}(t; U)$ yields either a guarded tree or a tree starting with $\emptyset$, a node, or a sync. Hence, $\mathit{ileave}(t; U)$ has a form for which we have already discussed the result of *exec*. To recap, to interleave $U = \{t_1, \ldots, t_k\}$, choose a t_i and replace it by $\mathit{ileave}(t_i; U - t_i)$.

DEFINITION 3.3 *(Non-Deterministic Interleaving)*

The non-deterministic interleaving of $\mathsf{par}(t_1, \ldots, t_k)$ is

$$\mathsf{ext}(\emptyset; \mathit{ileave}(t_1; U - t_1), \ldots, \mathit{ileave}(t_k; U - t_k))$$


t_1
...
t_k
$\mapsto$
$\mathit{ileave}(t_1;\, U - t_1)$
...
$\mathit{ileave}(t_k;\, U - t_k)$

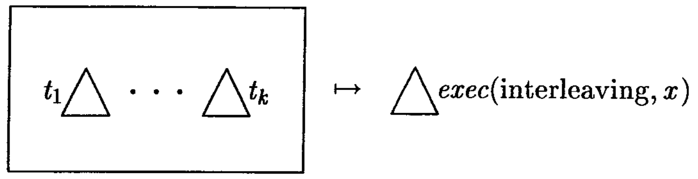
where $U = \{t_1, \ldots, t_k\}$.

□

Using this formula, it is straightforward to define *exec* for the remaining two forms of trees: replace the par by its non-deterministic interleaving, and apply *exec* to the resulting tree, with the same initial state.

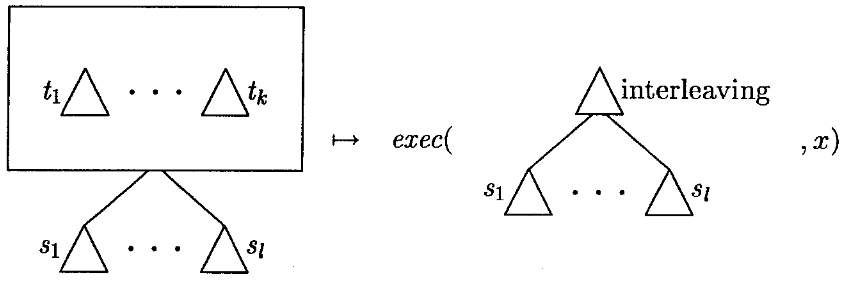
- $\mathsf{par}(t_1, \ldots, t_k)$
  Let bag $U = \{t_1, \ldots, t_k\}$.

E7 $$\mathit{exec}(\mathsf{par}(t_1, \ldots, t_k), x) = \mathit{exec}(\mathsf{ext}(\emptyset; \mathit{ileave}(t_1; U - t_1), \ldots, \mathit{ileave}(t_k; U - t_k)), x)$$



- $\mathsf{ext}(\mathsf{par}(t_1, \ldots, t_k); S)$
  Let bag $U = \{t_1, \ldots, t_k\}$.

E8 $$\mathit{exec}(\mathsf{ext}(\mathsf{par}(t_1, \ldots, t_k); S), x) = \mathit{exec}(\mathsf{ext}(\mathsf{ext}(\emptyset; \mathit{ileave}(t_1; U - t_1), \ldots, \mathit{ileave}(t_k; U - t_k)); S), x)$$



Examples of interleaving are given in the next section.

For easier reference, we repeat the complete definition of *exec*.

Let $x \in State$; $f, g \in State \to State$; $e \in State \to Boolean$; $t, t_1, \ldots \in Program$; $S \in$ **set of** $(Program)$; and $\ell \in Label$.

E1. $exec(\emptyset, x) = \emptyset$

E2. $exec(\mathsf{ext}(\emptyset; t_1, \ldots, t_k), x) = \mathsf{ext}(\emptyset; exec(t_1, x), \ldots, exec(t_k, x))$

E3. $exec(f, x) = f(x)$

E4. $exec(\mathsf{ext}(f; t_1, \ldots, t_k), x) = \mathsf{ext}(f(x); exec(t_1, f(x)), \ldots, exec(t_k, f(x)))$

E5. $exec(e \to t, x) = \begin{cases} \textbf{true} \to exec(t, x) & \text{if } e(x) \\ \textbf{false} \to \emptyset & \text{if } \neg e(x) \end{cases}$

E6. $exec(\mathsf{ext}(\mathsf{sync}; t_1, \ldots, t_k), x) = exec(\mathsf{sync}, x) = \textbf{false} \to \emptyset$,
where sync is $\mathsf{sync}_\ell(f; g)$.

E7. $exec(\mathsf{par}(t_1, \ldots, t_k), x) = exec(\mathsf{ext}(\emptyset; ileave(t_1; U - t_1), \ldots, ileave(t_k; U - t_k)), x)$
where bag $U = \{t_1, \ldots, t_k\}$.

E8. $exec(\mathsf{ext}(\mathsf{par}(t_1, \ldots, t_k); S), x) =$
$exec(\mathsf{ext}(\mathsf{ext}(\emptyset; ileave(t_1; U - t_1), \ldots, ileave(t_k; U - t_k)); S), x)$

where bag $U = \{t_1, \ldots, t_k\}$.

From the above definition it follows that an execution tree $exec(t, x)$ has no sync nodes and no par constructs. Furthermore, the only tree of the form **false** $\to t$ is **false** $\to \emptyset$. Next we define a relation, written '$\in$,' between traces and execution trees: If p is a trace and t an execution tree, $p \in t$ means that p is the sequence of states on a path through the tree. As discussed, $\emptyset$ nodes should not contribute to p, and **false** edges cannot be passed. In the definition, assume the tree is in normal form.

Let $Tree =$ **tree of** $(State, Boolean)$; $x \in State$; $p \in Trace$; $t \in Tree$; $S \in$ **set of** $(Tree)$.

e1. $\emptyset \in \emptyset$

e2. $x \in x$

e3. $p \in t \Rightarrow p \in \textbf{true} \to t$

e4. $\perp \in \textbf{false} \to \emptyset$

e5. $(p \in t \wedge t \neq \textbf{false} \to \emptyset) \Rightarrow n +\!\!+ p \in \mathsf{ext}(n; S \cup t)$, where n is x or $\emptyset$.

e6. $n +\!\!+ \perp \in \mathsf{ext}(n; \textbf{false} \to \emptyset)$, where n is x or $\emptyset$.

No node in the execution tree can have more than one outgoing edge labeled **false**, because then that node would have two children of the form **false** $\to \emptyset$, and by definition all children of a node must be different (because they form a set).

DEFINITION 3.4 *(Associated Trace-Set)*

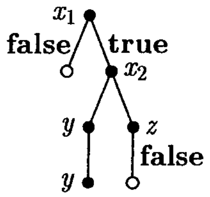
If X has the form of an execution tree, then its associated trace-set is the set

$$\left\langle \bigcup p \;:\; p \in \mathit{Trace} \wedge p \in X \;:\; p \right\rangle$$

□

We will use $exec(t,x)$ to denote the execution tree as well as its associated trace-set.

EXAMPLE 3.5

The associated trace-set of  is $\{x_1 +\!\!+ x_2 +\!\!+ y, \quad x_1 +\!\!+ x_2 +\!\!+ z +\!\!+ \perp\}$

(We have used the property that $y +\!\!+ y = y$.)

□

As we have seen in earlier examples, the execution tree $exec(t,x)$ need not start with the initial state x: If t starts with node f, then $exec(t,x)$ starts with $f(x)$ rather than with x. However, generally it is convenient to have the initial state as part of the computations of a program. Hence the following definition, which formalizes the concept of program execution.

DEFINITION 3.6 *(Program Execution)*

The execution of program t from initial state x yields the set of computations associated with the following execution tree:

$$Exec(t,x) = \mathsf{ext}(x; exec(t,x))$$

□

As with *exec*, $Exec(t,x)$ denotes both the execution tree and the associated trace-set.

3.2 Interleaving

In the previous section we defined the non-deterministic interleaving of $\mathsf{par}(t_1,\ldots,t_k)$ as

$$\mathsf{ext}(\emptyset; ileave(t_1; U - t_1), \ldots, ileave(t_k; U - t_k))$$

where $U = \{t_1,\ldots,t_k\}$. We specified that $ileave(t;U)$ is a tree starting with the first action (i.e., node or guard) of t, followed by a $\mathsf{par}(U,t')$ where t' is the remainder of t. In this section we give the precise definition of *ileave*.

It is clear that to define *ileave*, we must define what the first action and remainder of a tree are. If t is a single node, these definitions are obvious:

- The first action of f is f; the first action of $\emptyset$ is $\emptyset$; the first action of $\mathsf{sync}_\ell(f;g)$ is $\mathsf{sync}_\ell(f;g)$. In all three cases there is no remainder.

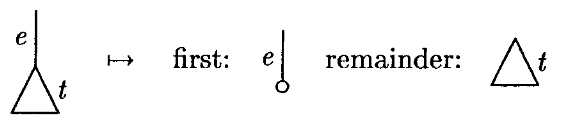
$$f\bullet \quad \mapsto \quad \text{first: } f\bullet \quad \text{no remainder}$$

$$\circ \quad \mapsto \quad \text{first: } \circ \quad \text{no remainder}$$

$$\times \quad \mapsto \quad \text{first: } \times \quad \text{no remainder}$$

If t is a guarded tree $e \longrightarrow t$, the first action is the guard:

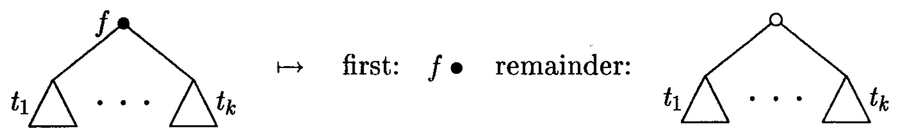
- The first action of $e \longrightarrow t$ is $e \longrightarrow \emptyset$, the remainder is t.

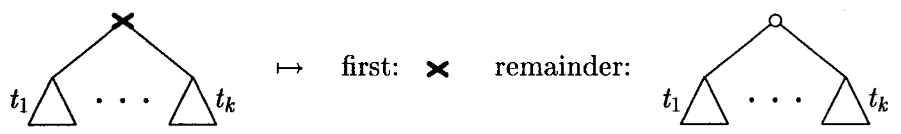


  ($e \longrightarrow \emptyset$ is used because e by itself is not a tree.)

If t has the form $\mathsf{ext}(f;t_1,\ldots,t_k)$ the first action is obviously f, but for the remainder we have the choice between a number of trees. Since we want the remainder to be a single tree, we add an $\emptyset$ node as place holder (if $k=1$, the $\emptyset$ node is removed again with axiom T7).

- The first action of $\mathsf{ext}(f;t_1,\ldots,t_k)$ is f; the first action of $\mathsf{ext}(\mathsf{sync};t_1,\ldots,t_k)$ is sync. In both cases, the remainder is $\mathsf{ext}(\emptyset;t_1,\ldots,t_k)$.





We cannot treat $t = \mathsf{ext}(\emptyset;t_1,\ldots,t_k)$ in the same way, because the remainder would be t again. In fact, the $\emptyset$ node is not a real action, because it has no effect on the execution. Rather, we should make a choice between $t_1,\ldots,t_k$ and take the first action and remainder of the selected tree. (In the case of $t = \emptyset$ we do say that $\emptyset$ is the first action, because there is simply no other choice.) A similar problem occurs with $t = \mathsf{par}(t_1, \ldots, t_k)$: there is more than one possible first node, and the remainder depends on which first node is chosen.

To simplify the definition of *ileave* in cases where there is such a choice between trees, we first define a function *choices* of type

$$choices : Program \rightarrow \textbf{set of}\,(Program)$$

choices(t) is the set of trees with which the execution can continue if t is reached. We define *choices* in such a way that each $t_i \in choices(t)$ has one of the forms mentioned above, for which the first action and remainder are well-defined.

If t has only one possible first action, then $choices(t) = \{t\}$.

For $t = \mathsf{ext}(\emptyset; t_1, \ldots, t_k)$ we have a choice between each of the subtrees. However, a subtree t_i may still have more than one choice of first node, namely if it starts with a par (it cannot start with $\emptyset$ because t is in normal form). Therefore, we define

$$choices(\mathsf{ext}(\emptyset; t_1, \ldots, t_k)) = choices(t_1) \cup \ldots \cup choices(t_k)$$

Suppose that, in an execution of $\mathsf{ext}(\emptyset; t_1, \ldots, t_k)$, tree t_1 is chosen; then, after the choice has been made, $t_2, \ldots, t_k$ are no longer relevant to the execution. This is in contrast with $\mathsf{par}(t_1, \ldots, t_k)$: if the first action of t_1 is chosen, $t_2, \ldots, t_k$ (plus the remainder of t_1) must still be executed. Hence, for $choices(\mathsf{par}(t_1, \ldots, t_k))$ we must make a choice between the first actions only, not simply between $t_1, \ldots, t_k$. Fortunately, we have already defined how this choice is made: replace the par construct by its interleaving. Hence,

$$choices(\mathsf{par}(t_1, \ldots, t_k)) = choices(\mathsf{ext}(\emptyset; ileave(t_1; U - t_1), \ldots, ileave(t_k; U - t_k)))$$

where $U = \{t_1, \ldots, t_k\}$.

For the last possible form of t, $\mathsf{ext}(\mathsf{par}(t_1, \ldots, t_k); S)$, we do of course exactly the same: replace the par by its interleaving. The complete definition of *choices* is summarized in the following table.

Let $f, g \in \mathit{State} \rightarrow \mathit{State}$; $e \in \mathit{State} \rightarrow \mathit{Boolean}$; $t, t_1, \ldots, t_k \in \mathit{Program}$; $S \in$ **set of** $(\mathit{Program})$; $\ell \in \mathit{Label}$.

c1. $\mathit{choices}(n) = \{n\}$ where n is $\emptyset$, f, or $\mathsf{sync}_\ell(f; g)$.

c2. $\mathit{choices}(e \rightarrow t) = \{e \rightarrow t\}$

c3. $\mathit{choices}(\mathsf{ext}(n; S)) = \{\mathsf{ext}(n; S)\}$ where n is f or $\mathsf{sync}_\ell(f; g)$, but not $\emptyset$.

c4. $\mathit{choices}(\mathsf{ext}(\emptyset; t_1, \ldots, t_k)) = \mathit{choices}(t_1) \cup \ldots \cup \mathit{choices}(t_k)$

c5. $\mathit{choices}(\mathsf{par}(t_1, \ldots, t_k)) = \mathit{choices}(\mathsf{ext}(\emptyset; \mathit{ileave}(t_1; U - t_1), \ldots, \mathit{ileave}(t_k; U - t_k)))$

where bag $U = \{t_1, \ldots, t_k\}$

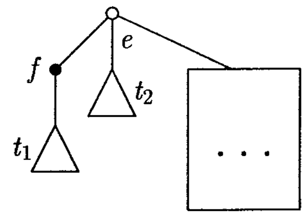
c6. $\mathit{choices}(\mathsf{ext}(\mathsf{par}(t_1, \ldots, t_k); S)) =$
$\mathit{choices}(\mathsf{ext}(\mathsf{ext}(\emptyset; \mathit{ileave}(t_1; U - t_1), \ldots, \mathit{ileave}(t_k; U - t_k)); S))$

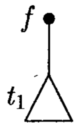
where bag $U = \{t_1, \ldots, t_k\}$

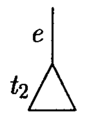
In some cases *choices* is recursive or involves *ileave* (which, as we will see in a moment, can involve *choices* again). But in these cases the argument of the recursive call is always a subtree of the original, so that the recursion ends as long as there is no infinite nesting of subtrees. The only infinite nesting of subtrees we use is of the form $\mathsf{ext}(f; \mathsf{ext}(f; \mathsf{ext}(f; \ldots)) \ldots)$, which is covered by c3 and does not require recursion to compute *choices*.

From the definition it follows that $t_i \in \mathit{choices}(t)$ does indeed have one of the forms for which the first node and remaining tree are well-defined. Once a $t_i \in \mathit{choices}(t)$ has been selected, the other choices are no longer relevant to the execution.

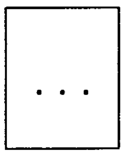
EXAMPLE 3.7

The *choices* of 

are 

, 

, and the

choices of 

.

□

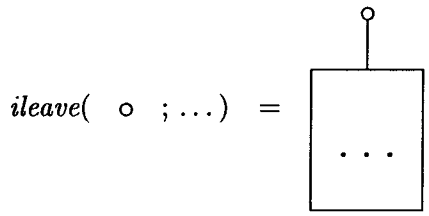
The main idea of $\mathit{ileave}(t; U)$ is to create a tree starting with the first action of t, followed by a par construct containing U and the remainder of t:

$$\mathit{ileave}(t; U) = \mathsf{ext}(\text{first action of } t; \mathsf{par}(\text{remainder of } t, U))$$

Hence, for the simple cases where there is only one choice of first action of t, we have the following (based on our earlier discussion of first action and remainder).

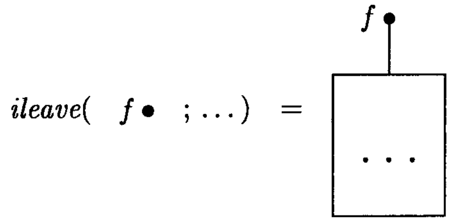
- $\emptyset$

I1 $$\mathit{ileave}(\emptyset; U) = \mathsf{ext}(\emptyset; \mathsf{par}(U))$$



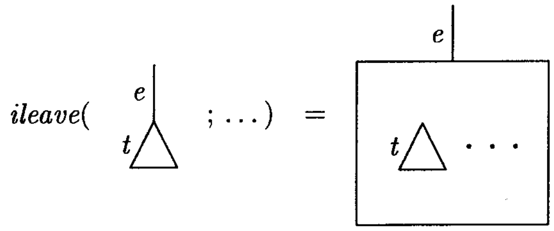
- f

I1 $$\mathit{ileave}(f; U) = \mathsf{ext}(f; \mathsf{par}(U))$$



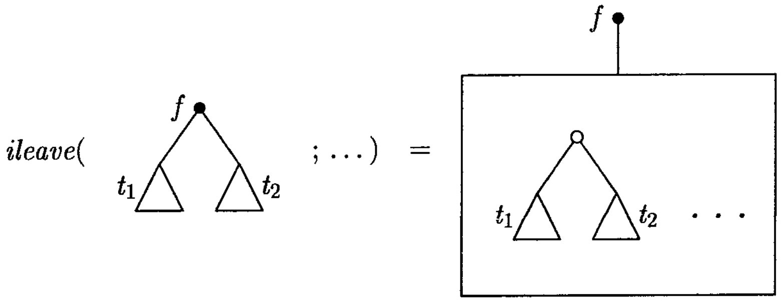
- $e \rightarrow t$

I2 $$\mathit{ileave}(e \rightarrow t; U) = e \rightarrow \mathsf{par}(t, U)$$



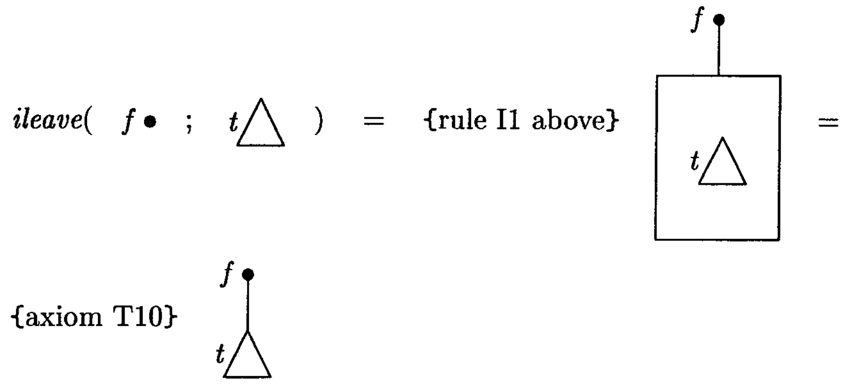
- $\mathsf{ext}(f; t_1, \ldots, t_k)$

I3 $$\mathit{ileave}(\mathsf{ext}(f; S); U) = \mathsf{ext}(f; \mathsf{par}(\mathsf{ext}(\emptyset; S), U))$$



Note that we have omitted trees of the forms sync and ext(sync;...), even though for these trees there is only one possible first action, namely the sync node. The reason is that sync nodes are interleaved in a special way, which we will get to in a moment.

EXAMPLE 3.8

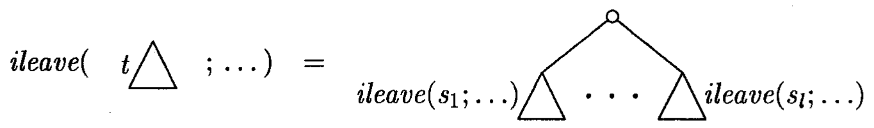


□

There are three types of trees t where there is a choice between possible first actions, as indicated by *choices*(t). Each $s_i \in$ *choices*(t) represents a possible first action of t together with the corresponding remainder of t. Hence, *ileave*$(s_i; U)$ is a possible value for *ileave*$(t; U)$. Since s_i has a form for which the first action and remainder are well-defined, *ileave*$(s_i; U)$ can be computed without further recursion. As usual, the choice between the s_i is made non-deterministically.

- ext($\emptyset$;...), par(...), ext(par(...);...)
  If t has one of these forms, then

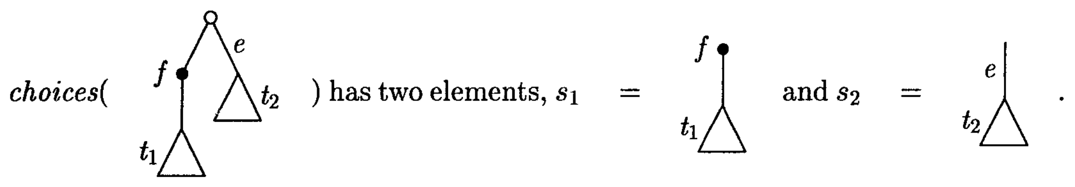
I5 $$\mathit{ileave}(t; U) = \mathsf{ext}(\emptyset; \mathit{ileave}(s_1; U), \ldots, \mathit{ileave}(s_l; U))$$

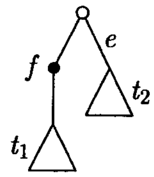


where $\{s_1, \ldots, s_l\} =$ *choices*(t).

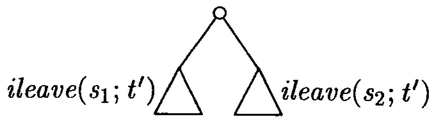
This last formula looks quite similar to the non-deterministic interleaving. However, if $t = \mathsf{par}(t_1, \ldots, t_k)$, the non-deterministic interleaving has subtrees *ileave*$(t_i; \ldots)$, whereas the above formula has subtrees *ileave*$(s_i; \ldots)$. Here, s_i includes the actions of all of $t_1, \ldots, t_k$, so $s_i \neq t_i$.

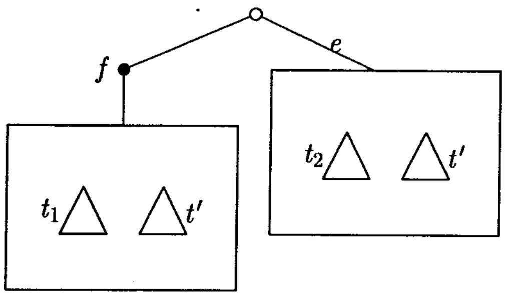
EXAMPLE 3.9



Therefore, *ileave*(

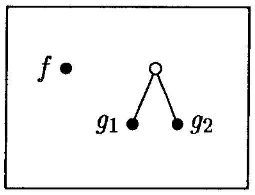
;

) = 

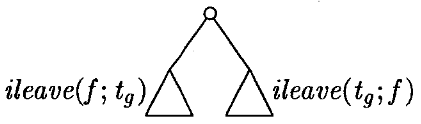
which is 

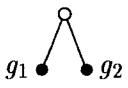
□

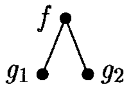
EXAMPLE 3.10

Let $t_g = \mathsf{ext}(\emptyset; g_1, g_2)$ and $t = \mathsf{par}(f, t_g) =$ 

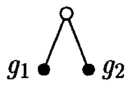
.

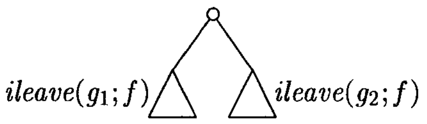
The non-deterministic interleaving of t is 

- The first interleaving is simple: *ileave*(f • ; 

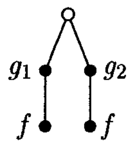
) = 

  (the $\emptyset$ node is removed using axiom T8).
- The second interleaving involves a choice between first actions, as in the previous example:

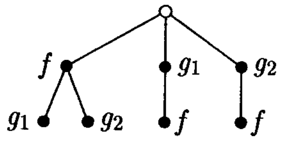
ileave(

; f •) = 

=



Hence, after combining the two interleavings and removing superfluous $\emptyset$ nodes (using T8), we find that the non-deterministic interleaving of t is



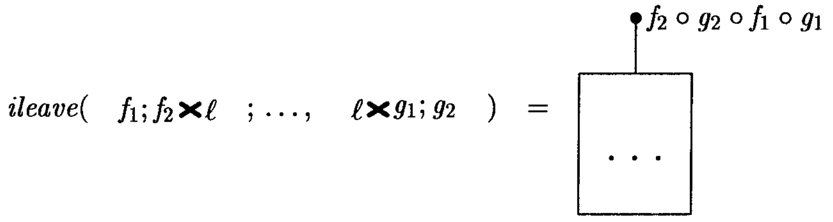
Usually the non-deterministic interleaving contains par constructs, but in this example these have the form par(s), which is equal to s (by axiom T10).

□

As already explained in Section 2.4, when two sync actions with identical ℓ labels are executed in parallel, they must be executed simultaneously in order to implement rendezvous style synchronization. Clearly, simultaneous execution of actions and interleaving of actions are contradictory techniques. Therefore, *ileave* treats sync nodes quite differently from normal nodes: Suppose the first action of t in *ileave*$(t; U)$ is $\mathsf{sync}_\ell(f_1; f_2)$. Then, if there is an $s \in U$ that has as a first action $\mathsf{sync}_\ell(g_1; g_2)$, with the same label ℓ, both sync actions are combined and replaced by their match; the match is then the first action of *ileave*$(t; U)$. Recall from Definition 2.15 that the match is the state transformer $f_2 \circ g_2 \circ f_1 \circ g_1$. Note that not only the first action of t is removed, but the first action of s as well.

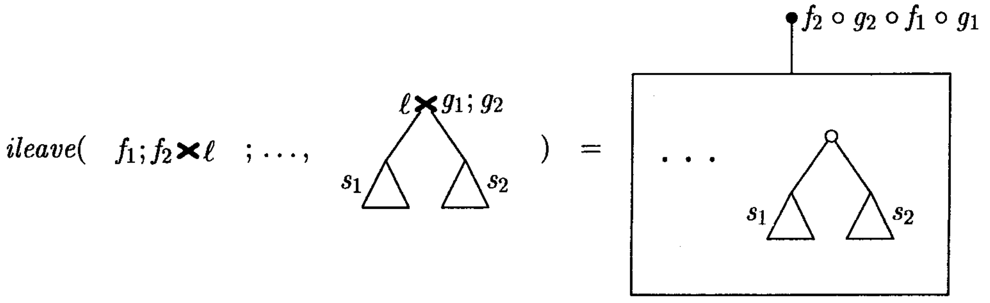
- $\mathsf{sync}_\ell(f_1; f_2)$
  Let $s \in U$ such that $\mathsf{sync}_\ell(g_1; g_2) \in$ *choices*(s).

I4 $$\textit{ileave}(\mathsf{sync}_\ell(f_1; f_2); U) = \mathsf{ext}(f_2 \circ g_2 \circ f_1 \circ g_1; \mathsf{par}(U - s))$$



- $\mathsf{sync}_\ell(f_1; f_2)$
  Let $s \in U$ such that $\mathsf{ext}(\mathsf{sync}_\ell(g_1; g_2); S) \in$ *choices*(s).

I4 $$\textit{ileave}(\mathsf{sync}_\ell(f_1; f_2); U) = \mathsf{ext}(f_2 \circ g_2 \circ f_1 \circ g_1; \mathsf{par}(U - s, \mathsf{ext}(\emptyset; S)))$$

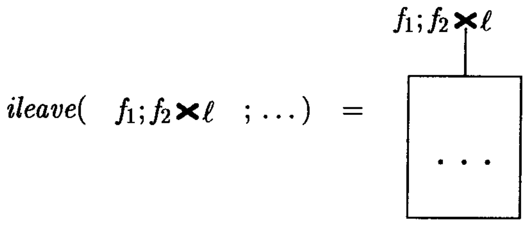


Of course, there are two more combinations, with t of the form $\mathsf{ext}(\mathsf{sync}_\ell(f_1;f_2);T)$; they are completely analogous.

If t starts with sync_ℓ and there is no $s \in U$ that has a matching sync_ℓ as first action, the sync node is treated as a normal node.

- $\mathsf{sync}_\ell(f_1;f_2)$
  Let there be no $s \in U$ such that $s' \in \mathit{choices}(s)$ starts with $\mathsf{sync}_\ell(g_1;g_2)$.

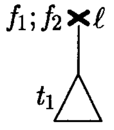
$$\text{I4} \qquad \mathit{ileave}(\mathsf{sync}_\ell(f_1;f_2);U) = \mathsf{ext}(\mathsf{sync}_\ell(f_1;f_2);\mathsf{par}(U))$$

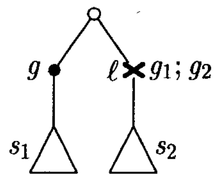


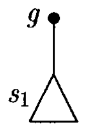
Again, the case with $t = \mathsf{ext}(\mathsf{sync}_\ell(f_1;f_2);T)$ is completely analogous.

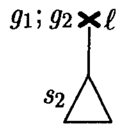
If there is more than one $s \in U$ that can start with a matching sync_ℓ, the above definition is ambiguous. Therefore, we require that there is at most one such s. In other words, we require that, during the execution, a sync node is either uniquely matched or not matched at all. In a programming language where synchronization is done over single-sender, single-receiver channels, as is the case in our version of CSP, this condition is already imposed by the programming language. Note that, by definition, *exec* is applied to trees in normal form. Therefore, the execution can never encounter a tree of the form $\mathsf{par}(\mathsf{par}(\mathsf{sync}_\ell,\mathsf{sync}_\ell),\mathsf{sync}_\ell)$; instead, the tree $\mathsf{par}(\mathsf{sync}_\ell,\mathsf{sync}_\ell,\mathsf{sync}_\ell)$ must be used. This latter tree violates the restriction that sync nodes must be uniquely matched. The restriction on unique matching could be removed by defining the interleaving to yield a non-deterministic choice between all possible matching pairs. However, that is unnecessarily complicated for our purposes.

EXAMPLE 3.11

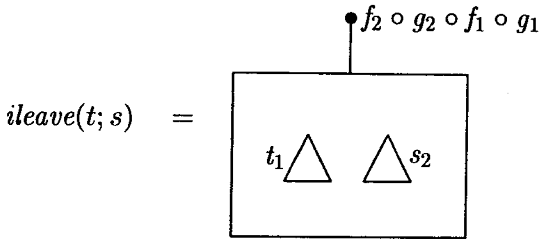
Let t =


and s =


. We want to compute *ileave*$(t;s)$. Since t starts with sync_ℓ, we check *choices*(s) to see whether s can start with a matching sync_ℓ. Since *choices*(s) has two elements,


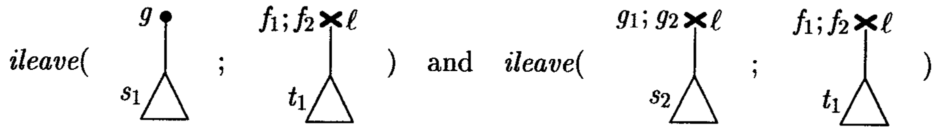
and


, this is indeed the case. Therefore

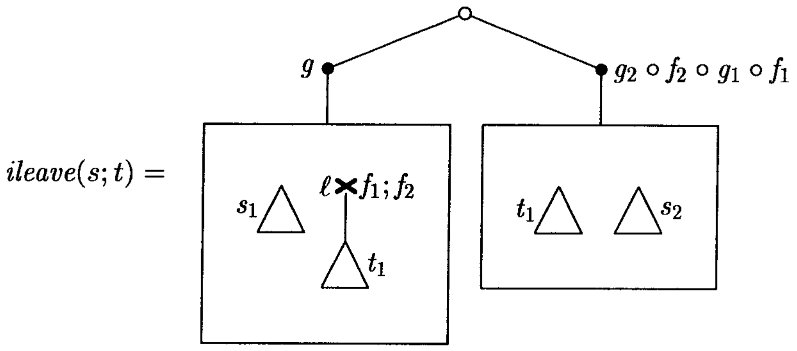


Note that the presence of the matched sync_ℓ nodes forces a choice in s: it is not possible to leave $\mathsf{sync}_\ell(f_1; f_2)$ unmatched and make the other choice in s.

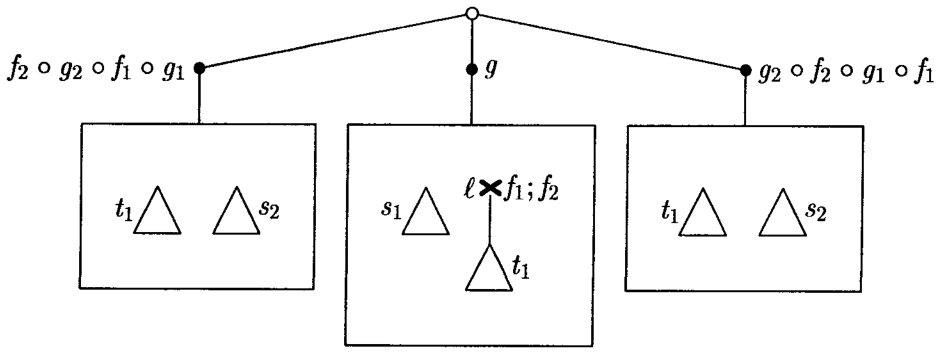
On the other hand, consider *ileave*$(s; t)$. Since *choices*(s) has two elements, we must compute two interleavings,



The result is



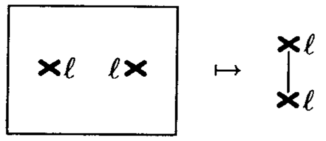
Combining *ileave*$(t; s)$ and *ileave*$(s; t)$, we find that the non-deterministic interleaving of $\mathsf{par}(t, s)$ is


$f_2 \circ g_2 \circ f_1 \circ g_1$
g
$g_2 \circ f_2 \circ g_1 \circ f_1$
t_1
s_2
s_1
$\ell \times f_1; f_2$
t_1
t_1
s_2

If we assume that the match is commutative (as it is for the constructs we have defined in Section 2.4), the left and right subtree are equal. In that case, the non-deterministic interleaving of par(t, s) happens to be equal to *ileave*$(s; t)$.

□

The *ileave* function would be easier if sync nodes could be interleaved as normal nodes, with the matching done in *exec*. However, that is not possible, because after interleaving there is not enough information left to perform the matching: If sync nodes were interleaved as normal nodes, the interleaving of par$(\mathsf{sync}_\ell, \mathsf{sync}_\ell)$ would be ext$(\mathsf{sync}_\ell; \mathsf{sync}_\ell)$:



In the *exec* function, these two sync_ℓ nodes would then have to be matched. But this interleaving cannot be distinguished from the sequential composition of two sync_ℓ nodes, which should not lead to a match.

We have now defined *ileave*$(t; U)$ for every form of t. The following table summarizes the definition.

Let $f, f_1, f_2, g_1, g_2 \in$ *State* $\to$ *State*; $e \in$ *State* $\to$ *Boolean*; $t, t_1, \ldots, t_k \in$ *Program*; $S, T \in$ **set of** (*Program*); $U \in$ **bag of** (*Program*).

I1. *ileave*$(n; U) = \mathsf{ext}(n; \mathsf{par}(U))$, where n is $\emptyset$ or f.

I2. *ileave*$(e \to t; U) = e \to \mathsf{par}(t, U)$

I3. *ileave*$(\mathsf{ext}(f; S); U) = \mathsf{ext}(f; \mathsf{par}(\mathsf{ext}(\emptyset; S), U))$

I4. Here, let $s \in U$ such that $\mathsf{sync}_\ell(g_1; g_2) \in$ *choices*(s) or $\mathsf{ext}(\mathsf{sync}_\ell(g_1; g_2); S) \in$ *choices*(s), if such an s exists. If there is a set S, let $s' = \mathsf{ext}(\emptyset; S)$; otherwise, $s' = \{\}$ (i.e., s' is the remainder of s). Let $m = f_2 \circ g_2 \circ f_1 \circ g_1$.

$$\left.\begin{array}{l} \mathit{ileave}(\mathsf{sync}_\ell(f_1; f_2); U) \\ \mathit{ileave}(\mathsf{ext}(\mathsf{sync}_\ell(f_1; f_2); T); U) \end{array}\right\} = \begin{cases} \mathsf{ext}(m; \mathsf{par}(t', U - s, s')) & \text{if } s \text{ exists} \\ \mathsf{ext}(\mathsf{sync}_\ell(f_1; f_2); \mathsf{par}(t', U)) & \text{otherwise} \end{cases}$$

where $t' = \mathsf{ext}(\emptyset; T)$ if there is a set T; otherwise $t' = \{\}$.
We require that there is at most one $s \in U$ with the given property. Also, the bag $t' \cup U - s \cup s'$ might be empty; in that case the par construct is omitted.

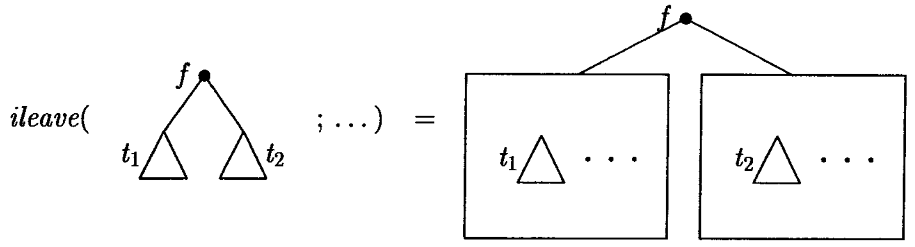
I5. In all other cases (i.e., t is $\mathsf{ext}(\emptyset; \ldots)$, $\mathsf{par}(\ldots)$, or $\mathsf{ext}(\mathsf{par}(\ldots); \ldots)$), the following rule applies:
ileave$(t; U) = \mathsf{ext}(\emptyset; \mathit{ileave}(s_1; U), \ldots, \mathit{ileave}(s_l; U))$
where $\{s_1, \ldots, s_l\} =$ *choices*(t).

We conclude the section with an example showing a subtlety of the interleaving.

EXAMPLE 3.12

At the beginning of the section, we decided that the remainder of $\mathsf{ext}(f; t_1, \ldots, t_k)$ was $\mathsf{ext}(\emptyset; t_1, \ldots, t_k)$, because we wanted it to be a single tree. However, we might consider having a set of remainders instead, and changing I3 to

I3 $\quad ileave(\mathsf{ext}(f; t_1, \ldots, t_k); U) \quad = \quad \mathsf{ext}(f; \mathsf{par}(t_1, U), \ldots, \mathsf{par}(t_k, U))$



This alternative definition may appear simpler and therefore preferable. However, suppose t_1 is a guarded tree, $e \rightarrow t_1'$. Then $\mathsf{par}(t_1, U)$ should only be executed if evaluation of e yields **true**. But since $\mathsf{par}(t_1, U)$ need not start with e, part of U can be executed before the guard is reached. Furthermore, even if e is **true** when the choice is made (i.e., immediately after execution of f), when the execution actually reaches e in $\mathsf{par}(t_1, U)$, e may be **false** and, for instance, lead to deadlock.

In short, this alternative definition is wrong because the choice between $t_1, \ldots, t_k$ is made before any guards are evaluated; hence the choice is made using demonic rather than guarded non-determinism. (If, however, none of $t_1, \ldots, t_k$ is guarded, both definitions are equivalent.)

□

3.3 Environments

At the beginning of Chapter 2 we remarked that we use the term program also to indicate a program part. This was later formalized by equating programs with trees, since a tree is built from smaller trees. The previous section described how a program can be executed from an initial state. However, most program parts are not intended to be executed in isolation, but only in combination with other program parts; these other parts together are called the *environment* of the program. This section gives a formal definition of environments.

DEFINITION 3.13 *(Tree Template)*

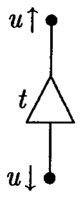
A tree template is a tree constructed with the standard constructors of the ADT, but which may contain unknown subtrees, i.e., subtrees that are only identified by name.

□

Hence, a tree template is a function from trees to trees, but not every function from trees to trees is a template.

EXAMPLE 3.14

Let t be an arbitrary (unknown) tree. Then

$\mathsf{ext}(u\uparrow; \mathsf{ext}(t; u\downarrow)) =$ 

is a tree template.

□

EXAMPLE 3.15

Function f_s defined by

$$f_s(t) = \begin{cases} \mathsf{ext}(n; t) & \text{if } t = s \\ e \to t & \text{if } t \neq s \end{cases}$$

is not a template, because there is no $\begin{cases} \dots & \text{if} \dots \\ \dots & \text{if} \dots \end{cases}$ constructor in the tree ADT.

□

DEFINITION 3.16 *(Environment)*

An environment is a function

$$\mathit{Program}, \mathit{Program}, \dots \to \mathit{Program}$$

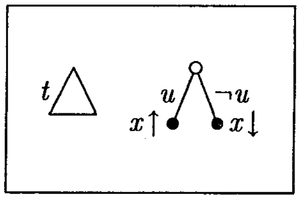
that can be described by a template.

□

An environment can have any number of parameters, but usually has just one or two. Say E is a one-parameter environment, and s and t are programs. We write $E[s]$ for the application of E to s. We often want to compare $E[s]$ with $E[t]$ by comparing s with t. However, this only makes sense if E uses s 'in the same way' as t. This is the reason to restrict environments to be templates, so that examples like f_s in Example 3.15 are excluded. This corresponds to saying that an environment can observe what a program does, but cannot observe the form of the program in any other way. Note that if E and F are one-parameter environments, then so is $E \circ F$.

EXAMPLE 3.17

Let t be an unknown tree. Then

$\mathsf{par}(t, \mathsf{ext}(\emptyset; u \to x\uparrow, \neg u \to x\downarrow)) =$  is a parallel environment for t. The value of u that is observed by the environment may depend on assignments made by t. Likewise, the behavior of t may depend on the environment's assignment to x.

The template of Example 3.14 is a sequential environment for t.

□

Most environments have restrictions on their parameters, i.e., not every program is in their domain. However, since we are usually interested in programs rather than environments, it is more convenient to express this as a property of programs, as follows. With $t \in \mathit{Program}$ we associate a set $\mathcal{V}(t)$ of *valid* environments of t, and a set $\mathcal{I}(t)$ of initial states for t. If $E \in \mathcal{V}(t)$, then $E[t]$ is a proper application of the environment, i.e., $t \in \mathcal{D}(E)$. Likewise, if $x \in \mathcal{I}(t)$, then $\mathit{Exec}(t,x)$ is a proper expression, i.e., x is in the domain of the function that maps state x to $\mathit{Exec}(t,x)$. Note that $\mathcal{V}(t)$ and $\mathcal{I}(t)$ are not functions of t: they are sets that can be more or less arbitrarily chosen. In particular, we usually restrict $\mathcal{V}(t)$ to exclude any environments we are not interested in. It follows that a complete program description is a triple $\langle t, \mathcal{V}(t), \mathcal{I}(t)\rangle$; t itself corresponds to the *code* of the program. However, we will continue to refer to t as the program, and mention any properties of $\mathcal{V}(t)$ and $\mathcal{I}(t)$ only when relevant. (This is similar to the common practice when discussing functions: a function is a pair $\langle f, \mathcal{D}(f)\rangle$, but usually f is also referred to as the function.)

DEFINITION 3.18 *(Valid Environments of $F[t]$)*

If $F \in \mathcal{V}(t)$, then $F[t]$ is a program, and hence has its own sets $\mathcal{V}(F[t])$ and $\mathcal{I}(F[t])$. However, we do not want to choose these sets arbitrarily; instead, we restrict them to have a certain relationship with $\mathcal{V}(t)$ and $\mathcal{I}(t)$.

Since $E[F[t]] = (E \circ F)[t]$, we require that

$$\mathcal{V}(F[t]) = \left\langle \bigcup E \;:\; E \circ F \in \mathcal{V}(t) \;:\; E \right\rangle$$

For $y \in \mathcal{I}(F[t])$ we require the following. Let t' stand for a tree starting with t (Definition 2.9). Suppose $\mathit{exec}(F[t], y)$ is evaluated by repeatedly using rules E1–E8. If that evaluation involves $\mathit{exec}(t', x)$ or $\mathit{exec}(\mathsf{par}(t', \ldots))$, then $x \in \mathcal{I}(t)$.

□

If E is a two-parameter environment, and we want to use $E[s,t]$, then it must be that $E[\bullet, t] \in \mathcal{V}(s)$ and $E[s, \bullet] \in \mathcal{V}(t)$ (where the $\bullet$ denotes a parameter).

DEFINITION 3.19 *(changes, depends, and var for Environments)*

For a one-parameter environment E we define

$$\mathit{changes}(E) = \left\langle \bigcap t \;:\; t \in \mathcal{D}(E) \;:\; \mathit{changes}(E[t]) \right\rangle$$

and likewise for $\mathit{depends}(E)$ and $\mathit{var}(E)$. The definition is extended in the obvious way for environments with more than one parameter.

□

DEFINITION 3.20 *(changes, depends, and var for Valid Sets)*

$$changes(\mathcal{V}(t)) = \left\langle \bigcup E \;:\; E \in \mathcal{V}(t) \;:\; changes(E[t]) \right\rangle$$

and likewise for $depends(\mathcal{V}(t))$ and $var(\mathcal{V}(t))$.

□

Hence, for instance, $var(E)$ consists of the variables that are always in $var(E[t])$ regardless of t.

3.4 The Implementation Relation

We are now ready to define what it means for a program s to implement a program t. We want 't is implemented by s' to mean that if t is used as part of some program, it can be replaced by s, and the difference will not be observable. As mentioned before, observations are made by environments, and what is observed are computations. Since we want to replace t by s, the environments in question are the environments of t, $\mathcal{V}(t)$. How do we know whether an environment observes a difference between s and t? It seems reasonable to say that if the environment observes a difference, then that should be reflected in the state of the environment. Therefore, when we compare a computation involving s with one involving t, we project the states in the trace onto the variables of the environment. This allows s and t to differ in the way they use their internal variables.

DEFINITION 3.21 *(Implementation)*

t is implemented by s means

$$\begin{array}{lll} t \leqslant s & \equiv & \mathcal{V}(t) \subseteq \mathcal{V}(s) \\ & & \wedge\; E \in \mathcal{V}(t) \;\Rightarrow\; \mathcal{I}(E[t]) \subseteq \mathcal{I}(E[s]) \\ & & \wedge\; \langle \forall E \;:\; E \in \mathcal{V}(t) \;: \\ & & \qquad \langle \forall x \;:\; x \in \mathcal{I}(E[t]) \;: \\ & & \qquad\qquad Exec(E[t], x) \lfloor var(E) \supseteq Exec(E[s], x) \lfloor var(E) \\ & & \quad \rangle \qquad \rangle \end{array}$$

□

The first two conjuncts are usually obvious, and we concentrate on the last one.

Note that there can be computations of $E[t]$ that cannot be generated by $E[s]$. But since the choice between computations of $E[t]$ is non-deterministic, the fact that $E[s]$ never makes some of the choices cannot be used to observe that s is not t. Hence, t is implemented by s, $t \leqslant s$, means that for all environments and initial states (of t), s behaves like t, but is possibly more deterministic. (Sometimes, $t \leqslant s$ is pronounced as 's is better than t.')

Equality between programs can be defined similarly, by replacing all $\subseteq$ and $\supseteq$ symbols with $=$ signs. We will write this as $t \equiv s$ to distinguish it from the $=$ operation on trees: $t \equiv s$ does certainly not mean that t and s are identical trees, i.e., that $t = s$ (it does not even mean that $Exec(E[t], x) = Exec(E[s], x)$). Although for most transformations from t to s in this text we have in fact $t \equiv s$, it may simplify proofs to prove just $t \leqslant s$.

Since $\subseteq$ is a partial ordering, it is straightforward to show that the implementation relation $\leqslant$ is a partial ordering as well. In particular, transitivity of $\leqslant$ means that complex transformations can be done (and proven) in several smaller steps. Another important property is that environments are monotonic with respect to $\leqslant$, i.e., that

$$t \leqslant s \Rightarrow F[t] \leqslant F[s]$$

assuming $F \in \mathcal{V}(t)$.

Proof: We prove each conjunct of Definition 3.21

1. $E \in \mathcal{V}(F[t]) \Rightarrow E \in \mathcal{V}(F[s])$:

$$\begin{aligned} & E \in \mathcal{V}(F[t]) \\ \equiv \quad & \{\text{Definition 3.18}\} \\ & E \circ F \in \mathcal{V}(t) \\ \Rightarrow \quad & \{s \geqslant t\} \\ & E \circ F \in \mathcal{V}(s) \\ \equiv \quad & \{F \in \mathcal{V}(s), \text{Definition 3.18}\} \\ & E \in \mathcal{V}(F[s]) \end{aligned}$$

2. $E \in \mathcal{V}(F[t]) \Rightarrow \mathcal{I}(E[F[t]]) \subseteq \mathcal{I}(E[F[s]])$:

$$\begin{aligned} & \mathcal{I}(E[F[t]]) \\ = \quad & \\ & \mathcal{I}((E \circ F)[t]) \\ \subseteq \quad & \{E \circ F \in \mathcal{V}(t) \wedge t \leqslant s\} \\ & \mathcal{I}((E \circ F)[s]) \\ = \quad & \\ & \mathcal{I}(E[F[s]]) \end{aligned}$$

3. Let $E \in \mathcal{V}(F[t])$ and $x \in \mathcal{I}(E[F[t]]) = \mathcal{I}((E \circ F)[t])$. Since $E \circ F \in \mathcal{V}(t)$ and $t \leqslant s$, it follows that

$$Exec((E \circ F)[t], x) \lfloor var(E \circ F) \supseteq Exec((E \circ F)[s], x) \lfloor var(E \circ F)$$

which is equivalent to

$$Exec(E[F[t]], x) \lfloor var(E \circ F) \supseteq Exec(E[F[s]], x) \lfloor var(E \circ F)$$

Since $var(E) \subseteq var(E \circ F)$, we know that, for trace p,

$$(p \lfloor var(E \circ F)) \lfloor var(E) = p \lfloor var(E)$$

(see Section 2.1). Another property of projection is that both sides of a $\supseteq$ symbol can be projected without changing the superset relation. Therefore, projecting both sides of the above equation on $var(E)$, we get

$$Exec(E[F[t]], x)\downharpoonright var(E) \supseteq Exec(E[F[s]], x)\downharpoonright var(E)$$

which proves the third conjunct.

□

This monotonicity property is called the *compositionality* of the implementation relation, or of the semantics.

THEOREM 3.22 *(Compositionality)*

For programs s and t, and $F \in \mathcal{V}(t)$:

$$t \leqslant s \Rightarrow F[t] \leqslant F[s]$$

□

Compositionality means that we can replace part of a program by an equivalent part, and get an equivalent program. Given the compositionality and transitivity of $\leqslant$ we can give proofs of the following form:

$$\begin{array}{ll} & F[t_1, t_2] \\ \leqslant & \{t_1 \leqslant s_1\} \\ & F[s_1, t_2] \\ \leqslant & \{t_2 \leqslant s_2\} \\ & F[s_1, s_2] \end{array}$$

from which we conclude, by transitivity, $F[t_1, t_2] \leqslant F[s_1, s_2]$.

This completes the definition of our semantic framework. See the remaining chapters of this text for examples of proofs of $t \leqslant s$.

Chapter 4
Analysis of Program Execution

In the previous two chapters we have defined what programs are and how they are executed. Program execution is important because it is the basis of the implementation relation. However, the *exec* function is rather complex, so that it is hard to see immediately what effect even a small change in a program has on the implementation relation. Therefore, in this chapter, we study the effect of such changes in general, to be used later when proving program transformations. In the first section we analyze the execution of a single program, and introduce some terms we use later. Following that there is a section each on the elementary changes to a tree: changing nodes, changing guards, and changing sync nodes.

We prove several small, but useful, program transformations in this chapter. Section 4.5 and Section 4.6 each describe a somewhat more complex transformation.

4.1 Observations about *exec*

Recall that if $t \in \mathit{Program}$ and $x \in \mathit{State}$ then $\mathit{exec}(t, x)$ is a tree of type **tree of** $(\mathit{State}, \mathit{Boolean})$; this tree, which we call the execution tree, is associated with a set of traces corresponding to the different possible computations. A natural way to study the effect of *exec* is to apply *exec* 'one step at a time,' where a step is an application of one of the rules E1–E8. We call this the *stepwise evaluation* of *exec*, or sometimes the stepwise execution of the program. Note that, given t in normal form, there is always exactly one rule that can be applied as first step in $\mathit{exec}(t, x)$.

We are interested in the way the execution tree is constructed by the steps of the execution. For instance, say we apply E4:

$$\mathit{exec}(\mathsf{ext}(f; t_1, \ldots, t_k), x) = \mathsf{ext}(f(x); \mathit{exec}(t_1, f(x)), \ldots, \mathit{exec}(t_k, f(x)))$$

Without evaluating the recursive calls, i.e., without performing any more steps, we can see that the execution tree has the form $\mathsf{ext}(f(x); \ldots)$. We say that execution of $\mathsf{ext}(f; \ldots)$ *contributes* (or adds) node $f(x)$ to the execution tree. Conversely, we say that node $f(x)$ in the execution tree *corresponds* to f in the argument of *exec*.

We consider what the origin is of each possible constructor in the execution tree.

1a. Empty nodes, $\emptyset$, can occur as a result of E1 and E2. In that case, the empty node corresponds to an empty node in the argument of *exec*. These empty nodes are of little consequence, since, by t3, they are not observable in the associated traces. Empty nodes can also occur in the form **false** $\rightarrow \emptyset$, resulting from E5 or E6. According to e1–e6, these empty nodes do not occur in the associated computations.

1b. Nodes that are states occur as a result of E3 or E4. In either case, a node $f(x)$ corresponds to a node f in the argument of *exec*. In addition, the initial state of the execution occurs as a node in the execution tree, because $\mathit{Exec}(t, x) = \mathsf{ext}(x; \ldots)$.

1c. sync nodes do not occur, because none of E1–E8 results in them.

2. ext constructors result from E2 or E4: an ext in the execution tree is the result of the execution of an ext. Additionally, the rule $\mathit{Exec}(t, x) = \mathsf{ext}(x; \ldots)$ (Definition 3.6) also adds an ext.

3. A tree of the form **true** $\rightarrow t$ results from rule E5. Hence, a **true** edge is the result of execution of a guard. The only possible tree with a **false** guard is **false** $\rightarrow \emptyset$, the result of the execution of a guard (E5) or of the execution of an unmatched sync node (E6).

4. par constructors do not occur in the execution tree.

Rules E7 and E8 do not directly add constructors to the execution tree; instead, they transform the argument of *exec*, so that one of the rules E1–E6 can be applied. For instance, according to point 1b above, a node in the execution tree corresponds to a node in the argument of *exec*. However, that argument may have been modified by the *ileave* function, through E7 or E8. Let us consider under which circumstances the *ileave* function results in a tree of the form f or $\mathsf{ext}(f; \ldots)$. We see that I1, I3, and I4 can result in such trees. In case of I1 and I3, f is already a node in the argument of *ileave*. Only I4 creates a new node, namely by matching two sync nodes from the original tree. Hence, a node $f(x)$ (a state) in the execution tree corresponds to a node f (a state transformer) in the program or to the match of two sync nodes in the program.

Likewise, *ileave* can only result in a tree $e \rightarrow t$ by application of I2, which requires that the guard e is already present in the program. Hence, a **true** edge in the execution tree corresponds to a guard in the program. *ileave* can result in trees $\mathsf{sync}(\ldots)$ or $\mathsf{ext}(\mathsf{sync}; \ldots)$ through I4, if there is no matching sync in the interleaving. In that case the sync node must already be present in the program. Therefore, a **false** edge in the execution tree corresponds to a guard in the program or to an unmatched sync in the program.

However, the result of *ileave* can have ext constructors that are not present in the program: I1, I4, and I5 (together with c5 and c6) create new ext constructors; rules E7 and E8 do the same. Hence, an ext constructor in the execution tree is the result of an ext constructor in the program or of the interleaving of a par constructor in the program. Of particular interest in this context are *choices* (Definition 2.14): A program usually has few, if any, unguarded choices. But the interleaving of a single par in the program can result in many unguarded choices, mostly through E7 and E8. Unguarded choices are important, because they can never lead to a $\perp$ in a trace.

From the above the following lemma follows.

LEMMA 4.1

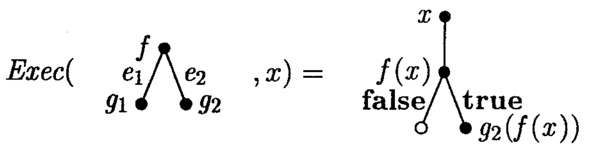
- Each node (state) in the execution tree corresponds either to a unique node (state transformer) in the program, or to a unique pair of matched sync nodes in the program, or to the initial state.
- Each labeled edge in the execution tree corresponds either to a unique guard in the program, or to a unique unmatched sync in the program.

□

The converse of this lemma is not true: not every program node or guard corresponds to a node or guard in the execution tree. For instance, if, in E5, $\neg e(x)$ holds, $exec(e \to t, x) =$ **false** $\to \emptyset$, so that state transformers and guards in t do not contribute to the execution tree. We say that the execution does not *reach* these state transformers and guards. An unmatched sync has the same effect as a **false** guard, according to E6, and can make parts of the tree unreachable. It is easy to verify that no other rule from E1–E8 and I1–I5 puts state transformers or guards out of reach of the execution.

EXAMPLE 4.2

Suppose $\neg e_1(f(x)) \wedge e_2(f(x))$. Then

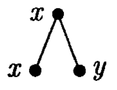


In the execution tree, x corresponds to the initial state; $f(x)$ corresponds to f; the **false** guard corresponds to e_1; the **true** guard corresponds to e_2; and $g_2(f(x))$ corresponds to g_2. Node g_1 of the program is out of reach of the execution, and does not contribute to the execution tree.

□

In the definition of the implements relation, the set of traces associated with $Exec(t, x)$ is used, rather than the execution tree itself. A trace in that set is a path through the execution tree, as described by rules e1–e6. Clearly, every state in a trace, except $\perp$, corresponds to a node in the execution tree. However, this node is not necessarily unique, because of the stuttering axiom for traces, t5.

EXAMPLE 4.3

The trace set associated with  has an element $x \mathbin{+\!\!+} x$, but this trace can also be written as just x; in this latter form (which is the normal form) it is unclear which of the two x nodes in the tree should be associated with state x in the trace.

In practice, which node we choose depends on which node we are interested in. For instance, we can say that the second x node contributes x to the trace, but that this state is not observable because it is preceded by an identical state.

Note that this phenomenon only occurs with sequences of identical states. Later we analyze when such sequences occur. In this example, y does not occur in trace $x +\!\!+ x$ at all; we say that y does not contribute to that trace.

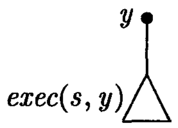
□

Next we derive a useful property of states traces. Suppose the stepwise evaluation of $exec(t, x)$ involves evaluation of $exec(s, y)$. Let $exec(s, y) = r$. Necessarily, r either is equal to $exec(t, x)$, or it is a proper subtree. If r is a subtree, then, according to T1–T4, r must occur as $\mathsf{ext}(r; \ldots)$, $\mathsf{ext}(n; r, \ldots)$, $e \rightarrow r$, or $\mathsf{par}(r, \ldots)$. But the first and last cases never occur during the evaluation of *exec*, and the third case only occurs as $\mathbf{true} \rightarrow r$. Hence, we have three possible cases:

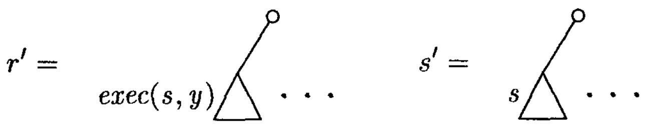
1. $r = exec(s, y) = exec(t, x)$.
2. $r = exec(s, y)$ occurs as $\mathsf{ext}(n; r, \ldots)$.
3. $r = exec(s, y)$ occurs as $\mathbf{true} \rightarrow r$.

We analyze each case.

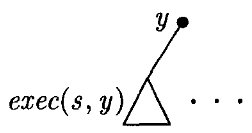
1. Since *exec* is only evaluated as part of *Exec*, if $exec(s, y)$ is not a proper subtree of some evaluation $exec(t, x)$, then it must be the evaluation occurring in $Exec(s, y) = \mathsf{ext}(y; exec(s, y))$. Hence, $exec(s, y)$ occurs in the execution tree as $\mathsf{ext}(y; exec(s, y))$:



2a. Such a subtree occurs as result of application of E2 or E4. If it results from E2, the execution tree has the form $r' = \mathsf{ext}(\emptyset; exec(s, y), \ldots)$. This tree corresponds to $s' = \mathsf{ext}(\emptyset; s, \ldots)$ (i.e., the left-hand side of E2), executed in the same state y. Hence, repeat the argument with $exec(s', y)$.

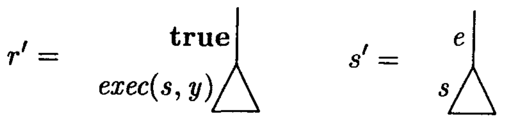


2b. If this case results from E4, then (in E4) $f(x) = y$, and the execution tree has the form $\mathsf{ext}(y; exec(s, y), \ldots)$:



3. An execution tree of the form $r' = \mathbf{true} \rightarrow exec(s, y)$ can only result from E5. Therefore, it corresponds to a tree $s' = e \rightarrow s$ executed in the same state y (for

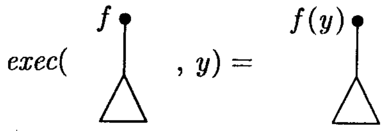
which $e(y)$ holds). Hence, repeat the argument with $exec(s', y)$.



Since the execution tree starts with case 1, cases 2a and 3 will always eventually lead to case 1 or 2b. Now consider the contribution to the traces of each case.

1. According to e5, the contribution of $exec(s, y)$ to a trace is immediately preceded by state y.
2a. According to e5 and t3, the contribution of $r = exec(s, y)$ to a trace is indistinguishable from the contribution of r'.
2b. This is the same as case 1.
3. According to e3, the contribution of $r = exec(s, y)$ to a trace is identical to the contribution of r'.

It follows that in a trace to which $exec(s, y)$ contributes, that contribution is immediately preceded by state y. In particular



which means that in the trace $f(y)$ is preceded by y. Because this is a useful result, we state it as a theorem.

THEOREM 4.4 *(Precondition Theorem)*
Let trace $p \in Exec(t, x)$.

- If $p = q \mathbin{+\!\!+} r$ where $r \in exec(s, y)$ is contributed to p by $exec(s, y)$, then r is immediately preceded by y: $p = q \mathbin{+\!\!+} y \mathbin{+\!\!+} r$.
- If $p = q \mathbin{+\!\!+} f(y) \mathbin{+\!\!+} r$, where $f(y)$ is contributed to p by state transformer f in state y, then that contribution is immediately preceded by y: $p = q \mathbin{+\!\!+} y \mathbin{+\!\!+} f(y) \mathbin{+\!\!+} r$.

□

Two remarks about this theorem: (1) If $q \mathbin{+\!\!+} r = q \mathbin{+\!\!+} y \mathbin{+\!\!+} r$, then either q ends in y or r starts with y, due to the stuttering axiom t5. (2) We say explicitly that $f(y)$ is contributed to the trace by f. This is because there may be another state transformer g which happens to contribute $g(z) = f(y)$; this contribution is preceded by z, not by y. Furthermore, we say explicitly that f was executed in state y. This is because there may be a z such that $f(z) = f(y)$, in which case $f(z)$ may be preceded by z rather than by y. Normally, when we write $f(y)$, we assume that this state was contributed by f in state y.

We mentioned before that a sequence $x \mathbin{+\!\!+} x$ in a trace is equal to just x, so that there is no unique state transformer that corresponds to x (Example 4.3). From the precondition theorem we see that a sequence $x \mathbin{+\!\!+} x$ occurs exactly when the second x is contributed by execution of f in state x with $f(x) = x$. Note however that the definition of the implementation relation has a projection on $var(E)$. Hence, the same phenomenon occurs in the projection under a weaker condition, namely when $f(x)\lfloor var(E) = x\lfloor var(E)$.

Next we define two well-known concepts in the context of our semantics.

DEFINITION 4.5 *(Pre- and Postcondition)*

- If f is a state transformer, and $exec(f, x)$ or $exec(\mathsf{ext}(f; \ldots), x)$ is evaluated (as part of a stepwise evaluation), then the set $pre(f) = \{x\}$ is called the precondition of f in that evaluation; the set $post(f) = \{f(x)\}$ is called the postcondition of f in that evaluation.
- If e is a guard, and $exec(e \rightarrow t, x)$ is evaluated, the precondition of e in that evaluation is the set $pre(e) = \{x\}$. The postcondition of e is defined as $post(e) = \langle \bigcup x : x \in pre(e) : \{x\}\rangle$.

□

Hence, the contribution of f to a trace is f's postcondition, and, by Theorem 4.4, is immediately preceded by f's precondition.

We have chosen singleton sets for the pre- and postconditions because of the following generalizations: If f occurs as part of a par, then $exec(f, x)$ can occur in the interleaving with different values of x. We define $pre(f)$ for that node f as the union of the preconditions in the different evaluations; likewise for $post(f)$. Furthermore, a tree $E[t]$ can contain many instances of node f. When $Exec(E[t], x)$ is evaluated, not all these instances need have the same precondition. We define the precondition of f in $Exec(E[t], x)$ as the union of all preconditions of instances of f in $E[t]$; likewise for $post(f)$. Generalizing further, $pre(f)$ in $E[t]$ stands for the union of all $pre(f)$ in $Exec(E[t], x)$, where the union is taken over all $x \in \mathcal{I}(E[t])$. Finally, $pre(f)$ in t stands for the union of all $pre(f)$ in $E[t]$, taken over all $E \in \mathcal{V}(t)$. Unless otherwise stated, $pre(f)$ refers to this last generalization, i.e., the union over all environments, initial states, and instances of f. In other words, $pre(f)$ is the set of all states in which f can be executed. $pre(e)$ and $post(e)$ are generalized in the same way.

We often use the abbreviation '$pre(f) \Rightarrow P$', defined as

$$pre(f) \Rightarrow P \quad \equiv \quad \langle \forall x : x \in pre(f) : P(x)\rangle$$

where P is a guard (a boolean expression on the variables of the state).

If $t = \mathsf{ext}(f; g)$, then if $exec(t, x)$ is evaluated, $pre(g) = post(f)$. However, if $t' = \mathsf{par}(t, s)$, then the same is not necessarily true for $exec(t', x)$, because the interleaving can produce trees like $\mathsf{ext}(f; \mathsf{ext}(h; g))$ where h is a node of s. In such cases, sometimes we can show that $pre(g)\lfloor A = post(f)\lfloor A$ for some set A, and then use $pre(g)\lfloor A \Rightarrow P$ instead of $pre(g) \Rightarrow P$. In this context, the concept of *interference* is significant.

DEFINITION 4.6 *(Conflicting Assignments)*

State transformers f and g are conflicting if

$$\langle \exists x, u \,:\, x \in State \wedge u \in changes(f) \wedge u \in changes(g) \,:\, (f(x))(u) \neq (g(x))(u) \rangle$$

□

For instance, $u\uparrow$ and $u\downarrow$ are conflicting assignments. The situation where conflicting assignments are interleaved is called *interference.*

DEFINITION 4.7 *(Interference)*

An execution $Exec(t, x)$ has interference if its stepwise evaluation involves $ileave(t; s, \ldots)$ where t starts with f and there is a tree in $choices(s)$ starting with g, such that f and g are conflicting.

□

This is generalized to programs in the obvious way: t has interference if there are $E \in \mathcal{V}(t)$ and $x \in \mathcal{I}(E[t])$ such that $Exec(E[t], x)$ has interference. In many cases, interference must be avoided.

A program t often has conflicting (but not interfering) assignments, for instance, $t = u\uparrow; \ldots; u\downarrow$. When environment E is applied to t, many occurrences of t can exist in $E[t]$; in particular, $E[t]$ can contain $\mathsf{par}(t, t)$. Since t has conflicting assignments, it is quite likely that $\mathsf{par}(t, t)$ has interference. For this reason, among others, such constructs must often be avoided.

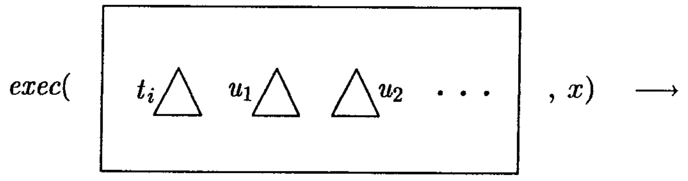
DEFINITION 4.8 *(Parallel Duplication)*

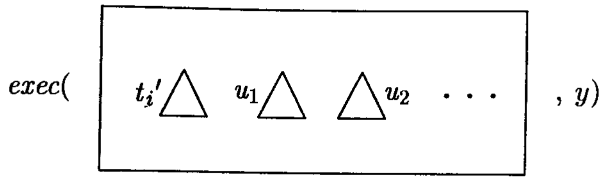
$E[t]$ has parallel duplication of t if, for any $x \in \mathcal{I}(E[t])$, the stepwise evaluation of $Exec(E[t], x)$ involves $ileave(t'; s, \ldots)$ where t' starts with a node of t and there is a tree in $choices(s)$ starting with a node of t.

□

Parallel duplication of t simply means that two instances of t are interleaved during an execution.

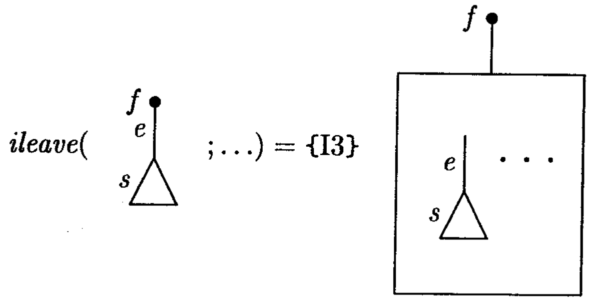
We conclude this section with some remarks about interleaving. Whenever $exec(\mathsf{par}(U'), x)$ is encountered, step E7 (or E8) requires evaluation of a number of interleavings of the form $ileave(t_i; U)$. If one of I1–I3 is applied, the result will be a tree to which one evaluation step of E1–E5 can be applied, but always of such a form that after this first step the execution again reaches a par construct, namely $exec(\mathsf{par}(U, t_i'), y)$. (This process can only terminate if $U \cup t_i'$ is a singleton, because then axiom T10 removes the par.) The important point is that, although t_i' differs from t_i (or maybe does not exist), the bag U is unchanged:



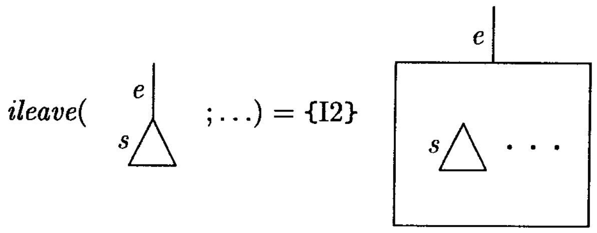


Rule I4, on the other hand, can change one of the trees in U, namely one that starts with a matched sync node. Even then, the other trees in U are not changed, and the changed tree is only affected to the extent that the sync node is removed.

Next, consider, as an example, $\mathit{ileave}(t; U)$ where $t = \mathsf{ext}(f; e \rightarrow s)$.



Hence, the first step of $\mathit{exec}(\mathit{ileave}(t; U), x)$ contributes $f(x)$ to the execution tree; this is exactly what $\mathit{exec}(t; x)$ would contribute. Let $t' = e \rightarrow s$, $x' = f(x)$. If we continue the execution (using E7), one of the possible choices is $\mathit{ileave}(t'; U)$.



Hence, $\mathit{exec}(\mathit{ileave}(t'; U), x')$ results in $\mathit{exec}(e \rightarrow \ldots, x')$, which, in the first step, again contributes the same as $\mathit{exec}(t', x')$ would contribute. It is straightforward to generalize this, leading to the following lemma.

LEMMA 4.9

For any t that does not start with a sync node matched by a tree in U, the contribution of the first step of $\mathit{exec}(\mathit{ileave}(t; U), x)$ is exactly the contribution of the first step of $\mathit{exec}(t, x)$.

If $t = \mathsf{ext}(\mathsf{sync}; S)$ where the sync node is matched in U, resulting in match f, the contribution of the first step of $\mathit{exec}(\mathit{ileave}(t; U), x)$ is exactly the contribution of the first step of $\mathit{exec}(\mathsf{ext}(f; S), x)$.

□

Going back to the example, $\mathit{ileave}(t'; U)$ is not the only possible choice; execution can also continue with $\mathit{ileave}(s; U - s, t')$. However, as we have seen above, such an interleaving leaves t' unchanged (if t' does not start with a matched sync). Hence, if the execution involves once again a node from t', it will be in the form $\mathit{exec}(\mathit{ileave}(t'; U'), y)$, which contributes in the same way as $\mathit{exec}(t', y)$. We see therefore that, if we restrict our attention to steps involving t, evaluation of $\mathit{exec}(\mathsf{par}(t, U), x)$ involves the 'same' steps as $\mathit{exec}(t, \ldots)$, except that the state can change between steps. This statement is not exact, however, because, due to the changing states, guards can evaluate to different values in both executions. Since a **false** guard removes part of the tree from consideration, we cannot quite say that both evaluations perform the same steps. But, keeping the changing states in mind, the analysis does show how the stepwise evaluation of $\mathit{exec}(\mathsf{par}(t, U), x)$ progresses with respect to t, namely similar to $\mathit{exec}(t, \ldots)$. We express this principle (the *interleaving principle*) by saying that interleaving maintains the execution order of individual trees.

4.2 Changing Nodes

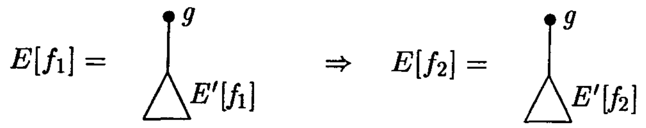
In the previous section we related the form of the execution tree $\mathit{Exec}(t, x)$ to the form of t. Our goal in this text is to prove transformations from tree t_1 into tree t_2 such that $t_1 \leqslant t_2$. Therefore, we have to relate the differences between two execution trees, $\mathit{Exec}(t_1, x)$ and $\mathit{Exec}(t_2, x)$, to the differences between t_1 and t_2. Obviously, large changes in a transformation are harder to analyze than simple changes. Therefore, in this section we consider the effect of changing just one node f_1 to f_2.

More precisely, let f_1 and f_2 be nodes, and $E \in \mathcal{V}(f_1)$ and $E \in \mathcal{V}(f_2)$ be an environment for both nodes. We want to compare $X_1 = \mathit{Exec}(E[f_1], x)$ with $X_2 = \mathit{Exec}(E[f_2], x)$, for some suitable initial state x. As before, we analyze the execution trees using stepwise evaluation. This time, we apply a step to each of the two trees we are comparing, then check what the differences between the resulting execution trees are. Whenever a step shows a difference between the two execution trees, we say that that step *exposes* a difference between X_1 and X_2.

The first step is the expansion of *Exec*, giving us $\mathsf{ext}(x; \mathit{exec}(E[f_1], x))$ and $\mathsf{ext}(x; \mathit{exec}(E[f_2], x))$. Without evaluating the recursive calls, we can only conclude that both execution trees have the form $\mathsf{ext}(x; \ldots)$, and hence that all traces of both executions start with x; therefore, this step does not expose any differences between X_1 and X_2.

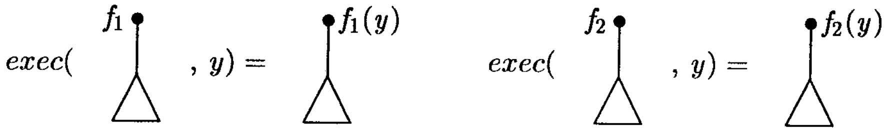
Now we make use of the fact that environment E must be a template. Therefore, the only difference between $E[f_1]$ and $E[f_2]$ is that a node f_1 in the first program can be replaced by f_2 in the second program. (It is possible that $E[t]$ has a node f_1 for every argument t, so that this particular node is not replaced by f_2. Generally, we only consider those nodes f_1 that are in fact replaced by f_2.) For instance, say $E[f_1] = \mathsf{ext}(g; E'[f_1])$

where $g \neq f_1$. Then necessarily $E[f_2] = \mathsf{ext}(g; E'[f_2])$.



In this case, only step E4 is applicable to both evaluations. The result, $\mathsf{ext}(g(x); \ldots)$, does not expose any differences between the execution trees. A similar argument applies to the other constructors, as well as to the recursive calls of *exec*.

Say X_1 involves an evaluation $exec(F[f_1], y)$ corresponding to $exec(F[f_2], y)$ in the evaluation of X_2. It is clear that at this point only steps E3 or E4 can expose a difference between the trees: E3 if $F[f_1] = f_1$ and $F[f_2] = f_2$, and E4 if $F[f_1] = \mathsf{ext}(f_1; \ldots)$ and $F[f_2] = \mathsf{ext}(f_2; \ldots)$. In either case, this step contributes $f_1(y)$ to X_1 and $f_2(y)$ to X_2, so that a difference is exposed unless $f_1(y) = f_2(y)$.



Because this difference involves the difference between f_1 and f_2 themselves, we call this a ***direct*** difference caused by the transformation from f_1 to f_2. The presence of a direct difference does not necessarily mean that the implementation relation does not hold between the trees we are comparing. In particular, a direct difference can only involve variables in $changes(f_1)$ or in $changes(f_2)$. If these variables are not in $var(E)$, then the direct difference between the traces is not observable after projection on $var(E)$.

EXAMPLE 4.10

Suppose we compare $Exec(E[\mathbf{skip}], x)$ with $Exec(E[u\uparrow], x)$. Then the direct difference is the difference between $\mathbf{skip}(y)$ and $u\uparrow(y)$ for some y, i.e., the difference between y and $y[u \mapsto \mathbf{true}]$.

$changes(\mathbf{skip}) = \{\}$ and $changes(u\uparrow) = \{u\}$. Therefore, if $u \notin var(E)$ then the direct difference may exist but is not observable after projection of the traces.

If, in $Exec(E[u\uparrow], x)$, we have $pre(u\uparrow) \Rightarrow u$, then $y[u \mapsto \mathbf{true}] = y$ and there is in fact no direct difference.

□

Continuing the analysis, suppose we evaluated $exec(\mathsf{ext}(f_1; F'[f_1]), y)$ and $exec(\mathsf{ext}(f_2; F'[f_2]), y)$, exposing a direct difference because $f_1(y) \neq f_2(y)$. Let $y_1 = f_1(y)$ and $y_2 = f_2(y)$. Since environments are templates, $F'[f_1]$ and $F'[f_2]$ once again have similar shapes, and the same step from E1–E8 applies to both trees. However, now the executions continue with different states: $exec(F'[f_1], y_1)$ versus $exec(F'[f_2], y_2)$. Since the states are different, each of E3, E4, and E5 can now expose a difference. For instance, if both $F'[f_1]$

and $F''[f_2]$ have the form $\mathbf{ext}(g;\ldots)$, application of rule E4 adds states $g(y_1)$ and $g(y_2)$ to X_1 and X_2, respectively. If both states are indeed different, we call this an *indirect* difference (because it does not directly involve the transformed nodes f_1 and f_2). Note that $g(y_1) \neq g(y_2)$ is only possible if y_1 and y_2 differ in a variable of $depends(g)$. Also, an indirect difference in a trace must be preceded by a direct difference, although the latter need not be observable after projection.

E3 exposes indirect differences in the same way as E4. Rule E5 can also expose an indirect difference, namely if $e(y_1) \neq e(y_2)$ for guard e. Say $e(y_1)$ but $\neg e(y_2)$. Then $exec(e \rightarrow t, y_1)$ involves the recursive call $exec(t, y_1)$ after step E3; but $exec(e \rightarrow t, y_2) = \mathbf{false} \rightarrow \emptyset$, and involves no further steps. Differences caused by guard evaluations are discussed in more detail in the next section.

EXAMPLE 4.11 *(Vacuous Assignment)*

Suppose in $Exec(E[\mathbf{skip}], x)$ we have $pre(\mathbf{skip}) \Rightarrow u$. Then there are no differences between $Exec(E[\mathbf{skip}], x)$ and $Exec(E[u\uparrow], x)$.

Proof: We have already seen, in Example 4.10, that there is no direct difference between the executions. Therefore, there can be no indirect differences either, so that both executions must be the same.

Such an assignment, which does not change the state, is called a *vacuous* assignment.

□

Define a *new* variable as follows.

DEFINITION 4.12 *(New Variable)*

A variable u is a new variable for program t if

$$u \notin var(t) \wedge u \notin var(\mathcal{V}(t))$$

A variable u is an unused variable for program t if

$$u \notin depends(t) \wedge u \notin depends(\mathcal{V}(t))$$

Hence, there may be assignments to an unused variable.

□

We use the expression "t has only new variables" to denote that $var(t) \cap var(\mathcal{V}(t)) = \{\}$.

EXAMPLE 4.13 *(New Variable Insertion)*

Let u be a new variable for $F[\mathbf{skip}]$. Then

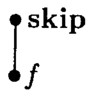
$$F[\mathbf{skip}] \equiv F[u\uparrow]$$

Proof: Take an arbitrary environment $E \in \mathcal{V}(F[\mathbf{skip}])$ and appropriate initial state x. Then compare $Exec(E[F[\mathbf{skip}]], x)$ with $Exec(E[F[u\uparrow]], x)$.

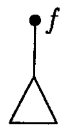
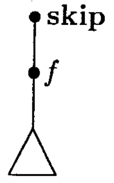
A direct difference caused by $u\uparrow$ can only involve variable u. But since $u \notin var(E)$, this direct difference is no longer observable after projection of the trace. Furthermore, since $u \notin var(E \circ F)$, there are no nodes or edges that depend on u, so there can be no indirect differences. Hence, after projection, both sets of traces are equal.

□

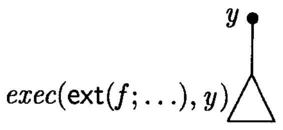
In the previous examples, we replaced a node **skip** by a different node. Such transformations are of little use unless there is a way to insert **skip** nodes in a program. Suppose we replace some node f by ext(**skip**; f).

$X_1 = Exec(E[\ \bullet f\ ], x) \qquad X_2 = Exec(E[$  $], x)$

The comparison of X_1 and X_2 is quite similar to the earlier comparison of the replacement of f_1 by f_2, except with respect to the direct difference. Say X_1 involves evaluation of $exec(F[f], y)$ corresponding to $exec(F[\text{ext}(\textbf{skip}; f)], y)$ in X_2. Once again, a difference can only be exposed if $F[f] = f$ and $F[\text{ext}(\textbf{skip}; f)] = \text{ext}(\textbf{skip}; f)$ or $F[f] = \text{ext}(f; \ldots)$ and $F[\text{ext}(\textbf{skip}; f)] = \text{ext}(\textbf{skip}; \text{ext}(f; \ldots))$. Say we have the latter situation (the other is similar).

$F[f] =$  $\qquad F[\text{ext}(\textbf{skip}; f)] =$ 

Instead of applying a step to both executions, we now apply E4 to the evaluation of X_2 only. This contributes state y to the tree, because $\textbf{skip}(y) = y$, and results in an execution tree of the form



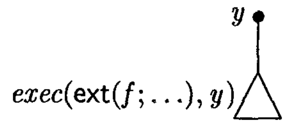
Following this step, both executions continue as before, both with execution trees of the form $exec(\text{ext}(f; \ldots), y)$. Hence, the direct difference is the extra node y in X_2. Since the state has not changed, there will be no indirect differences.

LEMMA 4.14 *(Skip Insertion)*

Let f be a node.

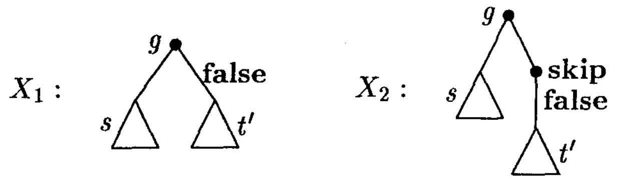
$$f \equiv ext(\textbf{skip}; f)$$

Proof: Let E be an appropriate environment and x an initial state. Compare $Exec(E[f], x)$ with $Exec(E[\text{ext}(\textbf{skip}; f)], x)$. From the analysis above it follows that the only differences between the execution trees are the extra states y in the second execution, occurring as

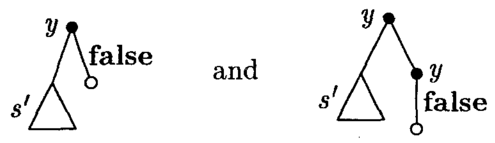


Hence, by e5, the traces of the second execution have an extra y immediately preceding the contribution $f(y)$ of f. But by the precondition theorem (Theorem 4.4), the traces of the first execution have the same state y preceding $f(y)$. Since $y \mathbin{+\!\!+} y = y$, the traces of both executions are identical.
□

Since $\mathsf{ext}(\mathbf{skip}; f)$ corresponds to the sequential composition $\mathbf{skip}; f$, this result is of course exactly what one would expect. However, in terms of the formal semantics, the result is not trivial. In particular, it hinges on the stuttering axiom for traces, t5, and on the precondition theorem; if we only used *exec* instead of *Exec*, the precondition theorem would not hold. Another subtlety is that we cannot change any arbitrary tree t to $\mathsf{ext}(\mathbf{skip}; t)$: Say $t = \mathbf{false} \rightarrow t'$, and the execution reaches $\mathsf{ext}(g; t, s)$ for X_1 and $\mathsf{ext}(g; \mathsf{ext}(\mathbf{skip}; t), s)$ for X_2.



Say g contributes state y. Then the execution trees will be



respectively. If s' does not have the form $\mathbf{false} \rightarrow \emptyset$, the $\mathbf{false}$ edge in X_1 will not contribute to any trace. But in X_2 there is the subtree $\mathsf{ext}(y; \mathbf{false} \rightarrow \emptyset)$, which corresponds to trace $y \mathbin{+\!\!+} \bot$. Although the extra y is not observable, the $\bot$ is, so that X_1 and X_2 are observably different. A similar situation occurs with insertion of **skip** before sync nodes. However, it is not hard to check that these are the only cases with this problem. Therefore, Lemma 4.14 can be generalized as follows.

LEMMA 4.15 *(Skip Insertion)*

For $f \in \mathit{State} \rightarrow \mathit{State}$; $t \in \mathit{Program}$; $U \in \mathbf{bag\,of}\ \mathit{Program}$:

$$f \equiv \mathsf{ext}(\mathbf{skip}; f)$$
$$t \equiv \mathsf{ext}(t; \mathbf{skip})$$
$$\mathsf{par}(t, U) \equiv \mathsf{par}(\mathsf{ext}(\mathbf{skip}; t), U)$$
$$\mathsf{par}(U) \equiv \mathsf{ext}(\mathbf{skip}; \mathsf{par}(U))$$

□

We now combine the **skip** insertion lemma with a slight generalization of Example 4.13, to get a very useful transformation.

THEOREM 4.16 *(State Variable Insertion)*

Let f be a node and $F[f]$ a program such that $u \notin \mathit{depends}(F[f])$ and $u \notin \mathit{var}(\mathcal{V}(F[f]))$. Then

$$F[f] \equiv F[u\uparrow; f]$$
$$F[f] \equiv F[u\downarrow; f]$$

(where $u\uparrow; f$ stands for $\mathsf{ext}(u\uparrow; f)$.)

Proof: We do the case with $u\uparrow$. First we apply the **skip** insertion lemma, giving us $F[f] \equiv F[\mathbf{skip}; f]$. Next we replace **skip** by $u\uparrow$. As in Example 4.13, the direct difference can only involve u. Since $u \notin \mathit{var}(E)$ for environment E, the direct difference is not observable. Furthermore, $u \notin \mathit{depends}(E[F[f]])$, so there can be no indirect differences.

□

Of course, the position of insertion of $u\uparrow$ can be generalized in the same way **skip** insertion was generalized.

The requirement that $u \notin \mathit{depends}(F[f])$ is weaker than requiring that u is a new variable. In particular, $u \notin \mathit{depends}(u\uparrow)$, so that the theorem can be used several times to insert $u\uparrow$ in different places, or to insert both $u\uparrow$ and $u\downarrow$.

State variable insertion is used in the proof of some more complex transformations, such as those discussed in Chapter 5. It is also used as a transformation by itself in order to distinguish between otherwise identical states in a computation; in this case the inserted variable is called a *state variable*. This transformation is performed prior to production rule expansion (see Chapter 6).

By now it should be clear that a proof of $t \leqslant s$ using stepwise evaluation follows a standard pattern. First we define $X_1 = \mathit{Exec}(E[t], x)$ and $X_2 = \mathit{Exec}(E[s], x)$. Then, using the fact that E is a template, we conclude that no difference can be exposed until X_1 evaluates $\mathit{exec}(F[t], y)$ corresponding to $\mathit{exec}(F[s], y)$, where $F[t] = t$ and $F[s] = s$, or $F[t] = \mathsf{ext}(t; \ldots)$ and $F[s] = \mathsf{ext}(s; \ldots)$. Up to this point of the proof, the difference between s and t has played no role whatsoever. Therefore, we simplify this preamble to the following.

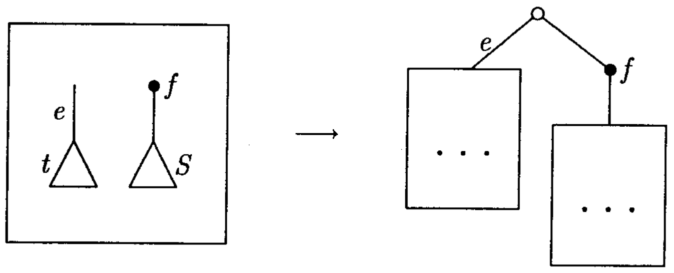
"Let X_1 (with t) and X_2 (with s) be the usual stepwise evaluations. Suppose t is reached in X_1, corresponding to s in X_2, both with state y."

This way, several fewer identifiers need to be introduced, no complex formulas need to be written, and we do not need to deal separately with both t and $\mathsf{ext}(t; \ldots)$. Following this preamble, we can immediately consider the differences between t and s. The goal is to show that any contribution to a trace by s can also be made by t. This usually means showing that any step in X_2 is either not observable (e.g., because of projection on $\mathit{var}(E)$), or can also be taken in X_1. Once both s and t have been passed in the executions, we can proceed immediately to the next point where s and t are reached. In addition to the simplified preamble, we will often use program notations rather than trees, following Section 2.4.

4.3 Changing Guards

Next we consider the effect of changing a guard. Let e_1 and e_2 be guards, t some tree, and $E \in \mathcal{V}(e_1 \rightarrow t)$ an environment for both $e_1 \rightarrow t$ and $e_2 \rightarrow t$. (We need t because a guard by itself is not a tree, and E takes a tree as argument.) We compare the usual stepwise evaluations X_1 (with $e_1 \rightarrow t$) and X_2 (with $e_2 \rightarrow t$).

Suppose $e_1 \rightarrow t$ is reached in X_1, and $e_2 \rightarrow t$ is reached in X_2, both with state y. This is the first point where a difference can be exposed, namely by E5. Clearly, if $e_1(y) = e_2(y)$, no difference will be exposed by this step (according to E5). But suppose $e_1(y) \wedge \neg e_2(y)$ holds. Then X_1 contains **true** $\rightarrow t'$ (for some t'), corresponding to **false** $\rightarrow \emptyset$ in X_2. According to e1–e6, a **true** edge never contributes anything to a trace. A **false** edge, on the other hand, can contribute $\bot$, but only in certain situations. If the **false** edge occurs as $\mathsf{ext}(y; \mathbf{false} \rightarrow \emptyset, t_1, \ldots, t_k)$ and at least one of the t_j is not **false** $\rightarrow \emptyset$, the **false** edge does not contribute $\bot$. In that case, the difference between X_1 and X_2 is that X_1 contains traces ending with contributions of t' which do not correspond to any traces in X_2. If our goal is to prove $e_1 \rightarrow t \leqslant e_2 \rightarrow t$, such differences are fine: we only need a subset relation, $X_2 \lfloor var(E) \subseteq X_1 \lfloor var(E)$, not equality. As we have pointed out before (in Section 4.1), if e_2 occurs in an unguarded choice, it can never contribute $\bot$. In particular, interleaving of $\mathsf{par}(e \rightarrow t, \mathsf{ext}(f; S))$ yields the unguarded choice $\mathsf{ext}(\emptyset; e \rightarrow \ldots, \mathsf{ext}(f; \ldots))$:



If, on the other hand, all t_j above have the form **false** $\rightarrow \emptyset$, a trace ending in $\bot$ is generated. Recall that a wait is a guard that appears outside a choice (Definition 2.14). Therefore, if e_2 is a wait and not part of a par construct, it will contribute $\bot$. This by itself does not mean that the implements relation does not hold, because X_1 may also have traces ending in $\bot$.

Naturally, the situation where $\neg e_1(y) \wedge e_2(y)$ is entirely symmetrical, except that the presence of extra traces is more of a problem when we want to prove that $e_1 \rightarrow t \leqslant e_2 \rightarrow t$.

Since a guard does not change the state, we do not distinguish between direct and indirect differences; instead, we have changes in the number of traces, and the possibility of $\perp$ insertion.

EXAMPLE 4.17 *(Vacuous Wait)*

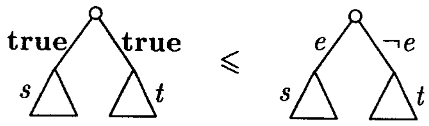
Suppose in $Exec(E[\mathbf{true} \rightarrow t], x)$ we have $pre(\mathbf{true}) \Rrightarrow d$. Then there are no differences between $Exec(E[\mathbf{true} \rightarrow t], x)$ and $Exec(E[d \rightarrow t], x)$.

Proof: In the analysis above we found that no difference is exposed whenever $e_1(y) = e_2(y)$.

Such a wait that always yields true is called a *vacuous* wait.

□

EXAMPLE 4.18



Proof: Since for any state y only one of $e(y)$ and $\neg e(y)$ can hold, the tree on the right-hand side contributes fewer traces than the tree with **true** guards (unless both trees do not contribute to any trace).

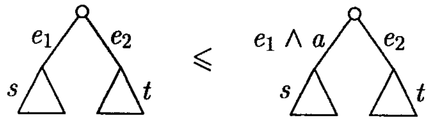
On the other hand, $e(y)$ and $\neg e(y)$ cannot both be false, so that no $\perp$ is inserted.

□

Instead of using expressions like $e_1(y) \wedge \neg e_2(y)$, we often refer to the value of $e_1 \wedge \neg e_2$ in state y.

LEMMA 4.19 *(Strengthening Guards)*

If $pre(e_1)$ implies that $e_1 \wedge \neg e_2 \Rrightarrow a$ then



Proof: Call the program on the left-hand side r_1, the other r_2. As before, r_2 may have fewer traces than r_1. Furthermore, r_2 can contribute $\perp$ if $\neg(e_1 \wedge a) \wedge \neg e_2$:

$$
\begin{array}{ll}
 & \neg(e_1 \wedge a) \wedge \neg e_2 \\
= & \\
 & (\neg e_1 \wedge \neg e_2) \vee (\neg a \wedge \neg e_2) \\
\Rrightarrow & \quad \{\neg a \Rrightarrow \neg e_1 \vee e_2\} \\
 & (\neg e_1 \wedge \neg e_2) \vee ((\neg e_1 \vee e_2) \wedge \neg e_2) \\
= & \\
 & \neg e_1 \wedge \neg e_2
\end{array}
$$

Hence, r_2 can only contribute $\perp$ if $\neg e_1 \wedge \neg e_2$, in which case r_1 also contributes $\perp$.

□

Guard strengthening is often done during production rule generation to make combinational gates. It is also used to make the guards of selection statements mutually exclusive (Theorem 6.12).

We can insert **true** guards in a program in the same way we can insert **skip** nodes. Suppose we replace node f by **true** $\rightarrow f$, and compare the usual X_1 and X_2. When f is reached in X_1, **true** $\rightarrow f$ is reached in X_2, say in the forms $\mathsf{ext}(f;\ldots)$ and **true** $\rightarrow \mathsf{ext}(f;\ldots)$. Suppose the state is y. We apply E5 to X_2 only, resulting in **true** $\rightarrow exec(\mathsf{ext}(f;\ldots), y)$. Following this step, both executions continue as before, both with execution of $\mathsf{ext}(f;\ldots)$ in state y. Therefore, the only difference between X_1 and X_2 is that $f(y)$ in X_1 is replaced by **true** $\rightarrow f(y)$ in X_2. According to e1–e6, the presence of the **true** label has no effect on the traces, so that the change from f to **true** $\rightarrow f$ is not observable.

As with **skip** node insertion, a **true** guard cannot be inserted before every tree. Since the same example suffices to illustrate this, we will not repeat the argument here. We conclude that **true** guards can be inserted in exactly those places where **skip** can be inserted.

LEMMA 4.20 *(True Guard Insertion)*

For $f \in State \rightarrow State$; $t \in Program$; $U \in$ **bag of** $Program$:

$$\begin{aligned} f &\equiv \mathbf{true} \rightarrow f \\ t &\equiv \mathsf{ext}(t; \mathbf{true} \rightarrow \emptyset) \\ \mathsf{par}(t, U) &\equiv \mathsf{par}(\mathbf{true} \rightarrow t, U) \\ \mathsf{par}(U) &\equiv \mathbf{true} \rightarrow \mathsf{par}(U) \end{aligned}$$

□

EXAMPLE 4.21

Suppose $u\uparrow$ does not occur within a par construct in $E[u\uparrow]$, for any $E \in \mathcal{V}(u\uparrow)$. Then

$$u\uparrow \equiv u\uparrow;\ [u]$$

where $u\uparrow;\ [u]$ stands for $\mathsf{ext}(u\uparrow; u \rightarrow \emptyset)$.

Proof: First we use Lemma 4.20 (the second clause), to get $u\uparrow \equiv u\uparrow;\ [\mathbf{true}]$. Since $u\uparrow$ does not occur within a par, $u\uparrow;\ [\mathbf{true}]$ is not involved in any interleaving, which allows us to conclude that $pre(\mathbf{true}) = post(u\uparrow)$. Hence, $pre(\mathbf{true}) \Rightarrow u$, and, as in Example 4.17, we can change **true** to u.

□

The restriction that $u\uparrow$ cannot occur within a par can be weakened a bit, as follows.

LEMMA 4.22 *(State Variable Wait Insertion)*

Let F be an environment of $u\uparrow$ such that $u\uparrow$ does not occur within a par construct. Suppose, for all $E \in \mathcal{V}(F[u\uparrow])$, there is no parallel duplication of $F[u\uparrow]$ in $E[F[u\uparrow]]$ and $u \notin var(E)$. Then

$$F[u\uparrow] \equiv F[u\uparrow;\ [u]]$$

Proof: From Lemma 4.20 we get $F[u\uparrow] \equiv F[u\uparrow;\ \texttt{[}\mathbf{true}\texttt{]}]$. Let E be an environment of $F[u\uparrow]$. If nodes are interleaved with $u\uparrow;\ \texttt{[}\mathit{true}\texttt{]}$, they must be nodes from E, because $F[u\uparrow]$ is not interleaved with itself and within $F[u\uparrow]$, $u\uparrow$ does not occur within a par construct. Since $u \notin \mathit{var}(E)$, we know that $\mathit{post}(u\uparrow)\lfloor\{u\} = \mathit{pre}(\mathbf{true})\lfloor\{u\}$. Hence, $\mathit{pre}(\mathbf{true}) \Rightarrow u$, and we can replace **true** by u.

□

Often, we first use the state variable insertion theorem (Theorem 4.16) to insert assignments $u\uparrow$ and $u\downarrow$ in a program F, then use the above lemma to insert appropriate waits $\texttt{[}u\texttt{]}$ and $\texttt{[}\neg u\texttt{]}$. It is straightforward to generalize the lemma to sequences like $u\uparrow; t;\ \texttt{[}u\texttt{]}$, where $u \notin \mathit{changes}(t)$.

We conclude this section with a lemma about insertion of **false** guards.

LEMMA 4.23 *(False Guard Insertion)*

For $f \in \mathit{State} \rightarrow \mathit{State}$; $t \in \mathit{Program}$; $S \in \mathbf{set\ of}\,(\mathit{Program})$:

$$\mathsf{ext}(f; S) \equiv \mathsf{ext}(f; S, \mathbf{false} \rightarrow t)$$

Proof: A difference between the execution trees can only be exposed when both trees are reached, say with state x. This yields execution trees $\mathsf{ext}(f(x); S')$ and $\mathsf{ext}(f(x); S', \mathbf{false} \rightarrow \emptyset)$, where S' depends on S and $f(x)$ only. This shows immediately that t does not contribute at all. Furthermore, the only way in which $\mathbf{false} \rightarrow \emptyset$ can be observed in a trace is if all trees in S' have the form $\mathbf{false} \rightarrow \emptyset$ as well; in that case, the trace of the second execution would end with $f(x) +\!\!+\ \bot$. But if all trees in S' are $\mathbf{false} \rightarrow \emptyset$, then the first execution of course also ends in $f(x) +\!\!+\ \bot$, so that no differences can be observed.

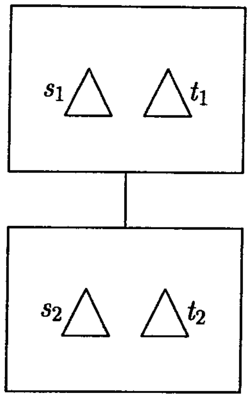
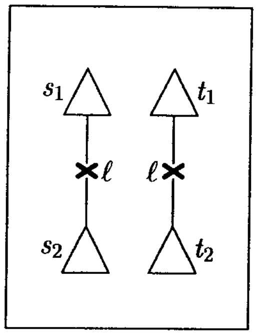
□

4.4 Changing Synchronization Nodes

If in some execution $\mathsf{sync}_\ell(f_1; f_2)$ is matched with $\mathsf{sync}_\ell(g_1; g_2)$, the actual contribution to the execution tree is the execution of the match $f_2 \circ g_2 \circ f_1 \circ g_1$. Hence, if we change any of the state transformers f_1, f_2, g_1, g_2, the effect is just that of changing the match. Since the match is a node, there is no need to analyze this type of change any further: it is covered by Section 4.2. If we change the state transformers of an unmatched sync, no effect is observable, since execution of unmatched sync nodes always results in $\mathbf{false} \rightarrow \emptyset$ (by rule E6).

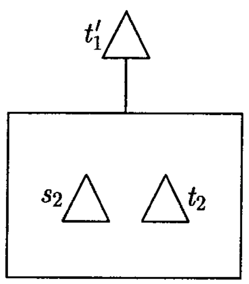
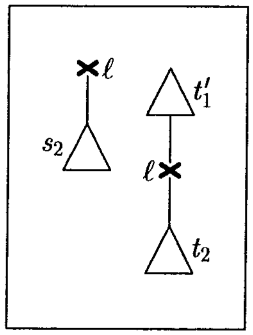
In practice, we seldom insert a single sync node, unless it is not reached during any execution. Because, if the new sync were matched during the execution, then it is likely that before the insertion there was an unmatched sync resulting in some trace ending in $\bot$. Not only would the $\bot$ no longer be generated, thus invalidating the implementation relationship, but also, programs that generate $\perp$ are usually considered erroneous; there is not much need to implement programs that are already wrong. If, on the other hand, the new sync is not matched, it has the same effect as insertion of a **false** guard. Practically the only case in which this is correct is the one expressed by the **false** guard insertion lemma, Lemma 4.23.

Therefore, we consider the effect of insertion of two sync nodes. Since we want the two nodes to be matched, they must naturally occur within a par construct. Consider

$r_1 =$  and $r_2 =$ 

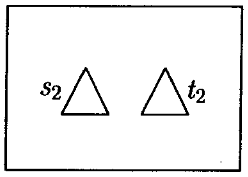
I.e., r_1 is $(s_1 \mathbin{//} t_1); (s_2 \mathbin{//} t_2)$ and r_2 is $(s_1; \mathsf{sync}_\ell; s_2) \mathbin{//} (t_1; \mathsf{sync}_\ell; t_2)$. Here, sync_ℓ stands for $\mathsf{sync}_\ell(\mathbf{skip}; \mathbf{skip})$.

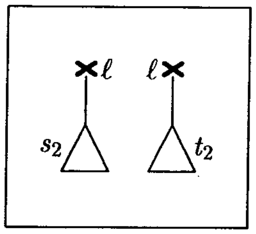
Compare r_1 and r_2, using the usual stepwise evaluations X_1 and X_2. Once r_1 and r_2 are reached, both evaluations start with a choice between two interleavings. X_1 has a choice between $\mathit{ileave}(s_1; t_1)$ and $\mathit{ileave}(t_1; s_1)$, and X_2 likewise has a choice between an interleaving starting with s_1 and one starting with t_1. Because interleaving maintains the order of execution (the interleaving principle), it is clear that no differences between X_1 and X_2 are exposed until X_1 has reached the end of s_1 or t_1. Say that the end of s_1 is reached first, with state y. Then the remaining evaluations have the form (X_1 is obtained by applying axiom T10)

X_1 :  and X_2 : 

Further evaluation of X_1 will contribute the same to the execution tree as $\mathit{exec}(t'_1, y)$ would, after which $\mathit{exec}(\mathsf{par}(s_2, t_2), z)$ remains. Evaluation of X_2, however, involves again an interleaving. There is a choice between a tree starting with sync_ℓ and a tree starting with the first action of t_1'. We assume that t_1' has no sync_ℓ node, so that the first choice contributes **false** $\to \emptyset$ (by E6). The second choice, however, contributes in the same way as $exec(t_1', y)$ would (Lemma 4.9). According to the **false** guard insertion lemma (Lemma 4.23), the **false** edge does not contribute to the trace. Therefore, the first step of both X_1 and X_2 is identical to the first step of $exec(t_1', y)$, and exposes no difference.

The same argument applies to all following steps, until either the sync_ℓ node in X_2 is matched, or the end of t_1' is reached. Since the sync_ℓ node can only be matched by a node following t_1', no differences are exposed until t_1' is finished. When that happens, X_1 and X_2 have reached

X_1 : 

and X_2 : 

both with the same state, say, z. Applying I4 to X_2 yields the (commutative) match **skip** $\circ$ **skip** $\circ$ **skip** $\circ$ **skip** $=$ **skip**. Hence, X_2 has reached $\mathsf{ext}(\mathbf{skip}; \mathsf{par}(s_2, t_2))$. According to the **skip** insertion lemma (Lemma 4.15), this is equivalent to $\mathsf{par}(s_2, t_2)$, so that both X_1 and X_2 continue with identical evaluations $exec(\mathsf{par}(s_2, t_2), z)$.

The above, admittedly somewhat complex, argument shows that sync nodes indeed *synchronize*. We summarize this in a theorem.

THEOREM 4.24 *(Synchronization Property)*

Let sync_ℓ stand for $\mathsf{sync}_\ell(\mathbf{skip}; \mathbf{skip})$. If $s_1 \mathbin{//} t_1$ does not contain any unmatched sync_ℓ node, and $\ell \notin var(E)$ for any appropriate environment, then

$$(s_1 \mathbin{//} t_1); (s_2 \mathbin{//} t_2) \equiv (s_1; \mathsf{sync}_\ell; s_2) \mathbin{//} (t_1; \mathsf{sync}_\ell; t_2)$$

Proof: Since $s_1 \mathbin{//} t_1$ does not contain unmatched sync_ℓ nodes, in the above argument we can indeed make the assumption that t_1' does not contain any sync_ℓ nodes. Furthermore, since $\ell \notin var(E)$, the interleaving is not affected by the environment (i.e., the inserted sync_ℓ nodes cannot be matched with a sync_ℓ in the environment). The above argument then proves the theorem.

□

Since the above argument is completely independent of s_2 and t_2, they can be omitted, giving us the following corollary.

COROLLARY 4.25

$$s_1 \mathbin{//} t_1 \equiv (s_1; \mathsf{sync}_\ell) \mathbin{//} (t_1; \mathsf{sync}_\ell)$$

□

EXAMPLE 4.26 *(Sync Duplication)*

Let sync stand for $\mathsf{sync}_\ell(\mathbf{skip};\mathbf{skip})$. Then

$$(s_1;\mathsf{sync};s_2)\ /\!/\ (t_1;\mathsf{sync};t_2) \equiv (s_1;\mathsf{sync};\mathsf{sync};s_2)\ /\!/\ (t_1;\mathsf{sync};\mathsf{sync};t_2)$$

Proof:

$$\begin{array}{ll} & (s_1;\mathsf{sync};s_2)\ /\!/\ (t_1;\mathsf{sync};t_2) \\ \equiv & \quad \{\text{synchronization property, Theorem 4.24}\} \\ & (s_1\ /\!/\ t_1);(s_2\ /\!/\ t_2) \\ \equiv & \quad \{\text{Corollary 4.25}\} \\ & ((s_1;\mathsf{sync})\ /\!/\ (t_1;\mathsf{sync}));(s_2\ /\!/\ t_2) \\ \equiv & \quad \{\text{synchronization property, Theorem 4.24}\} \\ & (s_1;\mathsf{sync};\mathsf{sync};s_2)\ /\!/\ (t_1;\mathsf{sync};\mathsf{sync};t_2) \end{array}$$

□

EXAMPLE 4.27

We can of course apply the synchronization theorem any number of times. In particular, we have the following ($\mathsf{sync} = \mathsf{sync}_\ell(\mathbf{skip};\mathbf{skip})$).

$$\begin{array}{ll} & *[s\ /\!/\ t] \\ \equiv & \\ & (s\ /\!/\ t);\ \ (s\ /\!/\ t);\ \ *[s\ /\!/\ t] \\ \equiv & \\ & (s;\mathsf{sync};s)\ /\!/\ (t;\mathsf{sync};t);\ \ *[s\ /\!/\ t] \\ \equiv & \\ & (s;\mathsf{sync};s)\ /\!/\ (t;\mathsf{sync};t);\ \ (s\ /\!/\ t);\ \ *[s\ /\!/\ t] \\ \equiv & \\ & (s;\mathsf{sync};s;\mathsf{sync};s)\ /\!/\ (t;\mathsf{sync};t;\mathsf{sync};t);\ \ *[s\ /\!/\ t] \\ \equiv & \\ & \qquad\quad \ldots \\ \equiv & \\ & *[s;\mathsf{sync}]\ /\!/\ *[t;\mathsf{sync}] \end{array}$$

□

4.5 Process Decomposition

In this section we discuss a transformation that is often performed at CSP level, called *process decomposition*. It is used to break up a process with a complex structure into two processes with simpler structures; the operation is akin to replacing a program part by a procedure call.

DEFINITION 4.28 *(Non-Terminating Process)*

If for all $f \in \mathit{State} \to \mathit{State}$

$$t \equiv t; f$$

then program t is called a non-terminating process.

□

We use 'process' instead of 'program' to avoid potential confusion between non-terminating processes and infinite programs: the latter are merely infinite trees. For instance, every loop is an infinite tree. If $t \equiv t; f$ for arbitrary f, this means that f never contributes to any trace. In practice, there are three situations in which f does not contribute to $\mathit{Exec}(E[t; f], x)$: (1) t itself is never reached in the evaluation, for instance because $E[t]$ contains **false** $\to t$; (2) evaluation of $\mathit{exec}(t, y)$ takes infinitely many steps; (3) t ends in deadlock, contributing **false** $\to \emptyset$ to the execution tree. In case (3), t contributes $\perp$ to the trace, unless there is always another choice of action available (in which case the trace is infinite).

EXAMPLE 4.29

For an arbitrary program t,

$*[\ t\ ]$

is an infinite process.

Proof: $*[t] = t; t; t; \ldots$. Hence, $*[t]$ either contributes infinitely many states to a trace, or contributes **false** $\to \emptyset$ to the execution tree. An infinite trace p followed by a contribution of f is just p, according to the trace axioms (t7, t8). If $*[t]$ contributes **false** $\to \emptyset$, then f is not reached and certainly does not contribute to any trace.

□

Recall that synchronization and communication actions can be defined with or without the possibility of probing. In the following we use two synchronization actions, $C?$ and $C!$, the first one of which can be probed. (No data is exchanged, the ? and ! are only used to distinguish between the two actions.)

$$C? = \mathsf{sync}_\ell(\mathbf{skip}; \mathbf{skip})$$
$$C! = \mathsf{ext}(\#C\!\uparrow; \mathsf{sync}_\ell(\mathbf{skip}; \#C\!\downarrow))$$

'$\#C$' is a boolean variable; we require that it is **false** in the initial state.

Consider a non-terminating process $F[t]$ for which $\#C$ and ℓ are new variables. Assume that for all $E \in \mathcal{V}(F[t])$ there is no parallel duplication of t in $E[F[t]]$. We transform $F[t]$ in two steps, starting with

$$F[t] \ \equiv\ (F[t] \,//\, *[[\#C \to t; C?]])$$

If we compare process decomposition with using procedures, this step corresponds to defining, but not calling, a procedure.

Proof: Let $r_1 = F[t]$, $s = *\texttt{[[}\#C \rightarrow t; C?\texttt{]]}$, and $r_2 = F[t] \mathbin{/\!/} s$. Consider the usual stepwise evaluations, X_1 with r_1 and X_2 with r_2, and assume r_1 and r_2 are reached.

In X_2, interleaving of r_2 yields a choice between two alternatives. One alternative has the form $\#C \rightarrow \ldots$, which evaluates to **false** $\rightarrow \emptyset$ ($\#C$ is initially false, and, since it is a new variable, no assignments to it are present). The other alternative starts with the first action of r_1, and therefore is identical to the first step of X_1. Hence, at this point of the evaluation, the only difference between X_1 and X_2 is the insertion of an alternative **false** $\rightarrow \emptyset$. According to the **false** guard insertion lemma (Lemma 4.23), this difference is not observable.

The same argument applies to all following steps. In each case, if X_1 has reached F', then X_2 has reached $F' \mathbin{/\!/} s$, and $\neg\#C$ holds. This process can only change if I1 can be applied to $\mathit{ileave}(F'; s)$, because that results in $\mathsf{par}(s) = s$ in X_2. Since then there is no longer a choice of actions, the **false** guard contributed by s can contribute $\perp$. However, if I1 can be applied to $\mathit{ileave}(F'; s)$, then apparently F' is a single node, and certainly the evaluation of $\mathit{ileave}(\mathsf{ext}(F'; f); s)$ can reach f in a finite number of steps. Since r_1 is a non-terminating process, this situation cannot occur (in a finite number of steps), so that no differences between X_1 and X_2 can be observed.

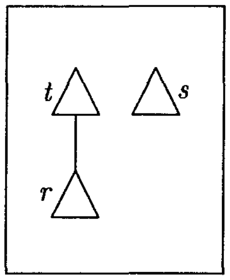
□

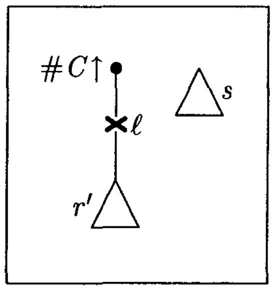
The second step of the transformation is

$$(F[t] \mathbin{/\!/} *\texttt{[[}\#C \rightarrow t; C?\texttt{]]}) \equiv (F[C!] \mathbin{/\!/} *\texttt{[[}\#C \rightarrow t; C?\texttt{]]})$$

In terms of procedures, this corresponds to replacing code (in this case t) by procedure calls.

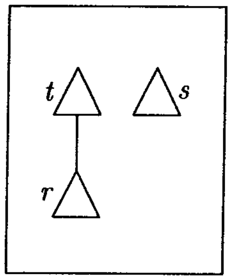
Proof: Compare the usual executions X_1 and X_2. A difference can first be exposed when evaluation of X_1 reaches t (in $F[t]$). This must occur as part of the evaluation of a par construct, because $F[t]$ occurs within a par. Note that $\neg\#C$ still holds at this point. Hence, suppose we have reached

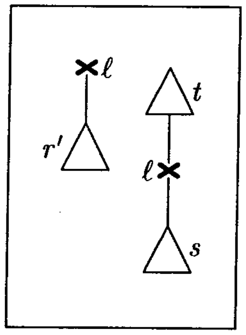
X_1 : 

X_2 : 

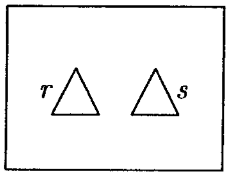
(where $s = *\texttt{[[}\#C \rightarrow t; C?\texttt{]]}$). In X_2, we have the choice between the **false** guard of s, which does not contribute to the trace, and the assignment $\#C\!\uparrow$; perform this assignment.

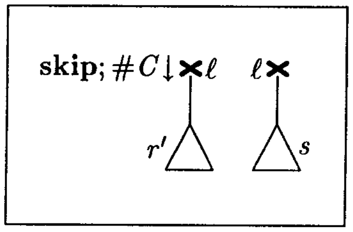
Because $\#C \notin var(E)$, no direct difference from $\#C\uparrow$ can be observed. For the next step of X_2, we have the choice between a sync and $\#C \to \ldots$. Since the sync is unmatched, it yields **false** $\to \emptyset$, which is not observable, because the other alternative yields **true** $\to \ldots$. The **true** edge does not contribute to the traces. Hence, following this step, no differences have yet been exposed, and the evaluations have reached

X_1 : 

X_2 : 

From the previous proof, we know that X_1 starts with the contributions of t, except that at each point there is an alternative **false** $\to \emptyset$. In X_2, there is the choice for an action of t, or for the sync$_\ell$ node. Since the sync$_\ell$ node is not matched, it contributes **false** $\to \emptyset$. This is true for every step until a matching sync$_\ell$ is reached, which only happens when t is finished. Hence, X_2 also starts with the contributions of t, with at each point an alternative **false** $\to \emptyset$. Therefore, X_1 and X_2 are identical until the end of t. At that point, the executions have reached

X_1 : 

X_2 : 

Applying I4 to X_2 yields the commutative match $\#C\downarrow$. As with $\#C\uparrow$, no direct difference from this assignment can be observed. Following $\#C\downarrow$, X_1 has reached par(r, s) and X_2 has reached par(r', s). If $r = F'[t]$ for some F', then $r' = F'[C!]$, and we are back to the starting situation (since $\neg\#C$ holds again).

□

We summarize this result in a theorem.

THEOREM 4.30 *(Process Decomposition)*

Let $F[t]$ be a non-terminating process for which $\#C$ and ℓ are new variables. Assume that for no $E \in \mathcal{V}(F[t])$ there is parallel duplication of t in $E[F[t]]$. Then

$$F[t] \;\equiv\; (F[C!] \;//\; *[[\#C \to t; C?]])$$

where $C! = \text{ext}(\#C\uparrow; \text{sync}_\ell(\textbf{skip}; \#C\downarrow))$ and $C? = \text{sync}_\ell(\textbf{skip}; \textbf{skip})$.

□

4.6 Process Factorization

Like process decomposition, process factorization transforms a single process into the parallel composition of two simpler processes. However, this transformation is normally performed at HSE level. Process factorization can be used as a transformation by itself, but is especially useful in the proofs of more complex transformations.

In this transformation we take a program of the form

$$t \;=\; [e_0];\ s_0;\ [e_1];\ s_1;\ \dots\ [e_j];\ s_j;\ \dots$$

where each e_j is a guard and each s_j a program. For e_j in this program, define condition C_j as

$$C_j = \langle \forall n \;:\; n \text{ is guard } e_{j-1} \vee n \text{ a node, guard, or sync in } s_{j-1} \;:\; \mathit{pre}(n) \Rightarrow \neg e_j \rangle$$

In words, C_j means that $\neg e_j$ holds when the wait $[e_{j-1}]$ is reached, and continues to hold during execution of s_{j-1}:

$$C_j = \quad [e_0];\ s_0;\ \dots\ s_{j-2};\ \underbrace{\underset{\uparrow}{[e_{j-1}]};\ s_{j-1};\ \underset{\uparrow}{\vphantom{x}}}_{\neg e_j}[e_j];\ \dots$$

We call C_j the *factorization condition* for e_j.

Assume that for every $j \geqslant 0$, C_j holds for all computations of t. Then we can implement this program by the parallel composition of two programs, t_{even} and t_{odd}:

$$t_{even} \;=\; [e_0];\ s_0;\ [e_2];\ s_2;\ \dots\ [e_{2j}];\ s_{2j}\ \dots$$

$$t_{odd} \;=\; [e_1];\ s_1;\ [e_3];\ s_3;\ \dots\ [e_{2j+1}];\ s_{2j+1}\ \dots$$

Note that C_0 is satisfied trivially; in fact e_0 can be just **true**. Also, C_1 implies that $\neg e_1$ holds initially.

Proof: Define t^j as the prefix of t ending with s_j; likewise for t^j_{even} and t^j_{odd}. The proof uses induction on j.
Basis:

$$t^0 \;\equiv\; t^0_{even}$$

This is trivially true, because both programs are identical, namely $[e_0]; s_0$.

Induction: There are two induction steps, one for extending t^j_{even} and one for extending t^j_{odd}. Both proofs are completely analogous; here we present the extension of t^j_{odd}. In this argument, the understanding is that if t^j does not exist because $j < 0$, then the execution of t^j is trivially finished; likewise for s_j. Also, $t^0_{even}\ /\!/\ t^{-1}_{odd}$ is just t^0_{even}.

Given the induction hypothesis (for $j \geqslant 0$)

$$t^{2j} \;\equiv\; t^{2j-2}_{even}; [e_{2j}]; s_{2j}\ /\!/\ t^{2j-1}_{odd}$$

prove that

$$t^{2j+1} \equiv t_{even}^{2j-2}; \texttt{[}e_{2j}\texttt{]}; s_{2j} \mathbin{//} t_{odd}^{2j-1}; \texttt{[}e_{2j+1}\texttt{]}; s_{2j+1}$$

(the latter is of course the same as $t^{2j+1} \equiv t_{even}^{2j} \mathbin{//} t_{odd}^{2j+1}$).

Consider the stepwise evaluations of the usual executions, X_1 (with t^{2j+1}) and X_2 (with the parallel construct). According to the induction hypothesis, no difference can be observed until the execution of t_{odd}^{2j-1} as part of X_2 is finished. That means that execution of s_{2j-1} is finished, both in X_1 and X_2. Because of the form of t^{2j}, if execution of s_{2j-1} is finished in X_1, then certainly execution of s_{2j-2} is finished in X_1. Since no difference has yet been observed, execution of s_{2j-2} must also be finished in X_2, meaning that execution of t_{even}^{2j-2} must be finished in X_2. We conclude then that the first difference can only be observed if the evaluation of X_1 has reached

$$\texttt{[}e_{2j}\texttt{]}; s_{2j}; \texttt{[}e_{2j+1}\texttt{]}; s_{2j+1}$$

and the evaluation of X_2 has reached

$$\texttt{[}e_{2j}\texttt{]}; s_{2j} \mathbin{//} \texttt{[}e_{2j+1}\texttt{]}; s_{2j+1}$$

In X_2 there is a choice between two interleavings, one of which has the form $e_{2j+1} \rightarrow \ldots$. Now, according to C_{2j+1}, we have $pre(e_{2j}) \Rightarrow \neg e_{2j+1}$. Since we have reached e_{2j} in X_1, without differences, we must indeed have $\neg e_{2j+1}$ in both X_1 and X_2. Therefore, in X_2, the interleaving starting with e_{2j+1} evaluates to **false** $\rightarrow \emptyset$. The other interleaving starts in the same way as the remaining program in X_1. Therefore, in this first step the only difference between X_1 and X_2 is an extra **false** guard, which, according to the **false** guard insertion lemma, is not observable. The same is true for all following steps, as long as $\neg e_{2j+1}$ holds. According to C_{2j+1}, this can only change after the execution of s_{2j}. Once execution of s_{2j} is finished, the parallel construct in X_2 reduces to just $\texttt{[}e_{2j+1}\texttt{]}; s_{2j+1}$ (by axiom T10). In X_1, once execution of s_{2j} is finished, exactly the same program remains. Hence, both executions are identical.

□

We summarize this result in a theorem.

THEOREM 4.31 *(Process Factorization)*

Define t, t_{even} and t_{odd} as

$$t = \texttt{[}e_0\texttt{]};\ s_0;\ \texttt{[}e_1\texttt{]};\ s_1;\ \ldots\ \texttt{[}e_j\texttt{]};\ s_j\ \ldots$$

$$t_{even} = \texttt{[}e_0\texttt{]};\ s_0;\ \texttt{[}e_2\texttt{]};\ s_2;\ \ldots\ \texttt{[}e_{2j}\texttt{]};\ s_{2j}\ \ldots$$

$$t_{odd} = \texttt{[}e_1\texttt{]};\ s_1;\ \texttt{[}e_3\texttt{]};\ s_3;\ \ldots\ \texttt{[}e_{2j+1}\texttt{]};\ s_{2j+1}\ \ldots$$

Define the factorization condition C_j for e_j in t as

$$C_j = \langle \forall n\ :\ n \text{ is guard } e_{j-1} \vee n \text{ a node, guard, or } \mathsf{sync} \text{ in } s_{j-1}\ :\ pre(n) \Rightarrow \neg e_j \rangle$$

If for all $j \geqslant 0$ the factorization condition C_j holds, then

$$t \equiv t_{even} \;//\; t_{odd}$$

□

Process factorization is usually applied to processes of the form $*[s]$. If $s \equiv s_{even} \;//\; s_{odd}$, then certainly $*[s] \equiv *[s_{even} \;//\; s_{odd}]$ (by Theorem 3.22). The following two examples show when the more interesting

$$*[s] \equiv *[s_{even}] \;//\; *[s_{odd}]$$

holds.

EXAMPLE 4.32

Let $t = *[s]$ be of the form

$$\begin{aligned} t &= *[[e_0];\ s_0;\ [e_1];\ \ldots\ [e_{2k-1}];\ s_{2k-1}\ ] \\ &= [e_0];\ s_0;\ [e_1];\ \ldots\ [e_{2k-1}];\ s_{2k-1};\ [e_0];\ s_0;\ [e_1];\ \ldots \end{aligned}$$

Hence, we can apply process factorization if C_{2k} holds for e_0; this results in

$$\begin{aligned} t_{even} &= [e_0];\ s_0;\ [e_2];\ \ldots\ [e_{2k-2}];\ s_{2k-2};\ [e_0];\ s_0;\ [e_2]\ \ldots \\ &= *[[e_0];\ s_0;\ [e_2];\ \ldots\ [e_{2k-2}];\ s_{2k-2}]] \end{aligned}$$

and

$$\begin{aligned} t_{odd} &= [e_1];\ s_1;\ [e_3];\ \ldots\ [e_{2k-1}];\ s_{2k-1};\ [e_1];\ s_1;\ [e_3]\ \ldots \\ &= *[[e_1];\ s_1;\ [e_3];\ \ldots\ [e_{2k-1}];\ s_{2k-1}]] \end{aligned}$$

It is important that t ends with an odd index, so that the second instance of e_0 corresponds to an even index and occurs in t_{even} instead of t_{odd}.

□

EXAMPLE 4.33

In the process factorization theorem, C_0 is trivially true, meaning that e_0 can be just **true**; in that case, by the **true** guard insertion lemma, $[e_0]$ can be omitted from t. The same can be done with a process of the form $*[s]$:

$$\begin{aligned} t &= *[\ s_0;\ [e_1];\ \ldots\ s_{2k-1};\ [e_{2k}]] \\ &= s_0;\ [e_1];\ \ldots\ s_{2k-1};\ [e_{2k}];\ s_0;\ [e_1];\ \ldots \end{aligned}$$

Process factorization can be applied if $pre(e_{2k}) \Rightarrow \neg e_1$, resulting in

$$\begin{aligned} t_{even} &= s_0;\ [e_2];\ \ldots\ s_{2k-2};\ [e_{2k}];\ s_0;\ [e_2];\ \ldots \\ &= *[\ s_0;\ [e_2];\ \ldots\ s_{2k-2};\ [e_{2k}]] \end{aligned}$$

and

$$\begin{aligned} t_{odd} &= [e_1];\ s_1;\ [e_3];\ \ldots\ s_{2k-1};\ [e_1];\ s_1;\ \ldots \\ &= *[[e_1];\ s_1;\ [e_3];\ \ldots\ s_{2k-1}\ ] \end{aligned}$$

□

Chapter 5
Synchronization

As explained in Section 1.2, the high-level design of a circuit is done in a CSP-like language. At some point of the design, CSP programs are converted to the more primitive HSE language. The main difference between CSP and HSE is that HSE has no synchronization actions. In this chapter we discuss transformations that replace synchronization actions by protocols using shared variables.

5.1 Handshake Expansion

In this section we study a transformation that replaces sync nodes with sequences of normal nodes (assignments) and waits; this transformation is called *handshake expansion*. The proof relies on the process factorization theorem, which was proven in the previous chapter. However, we start out by proving two lemmas about the behavior of programs when used as part of a larger program.

Consider a program t in an execution of $E[t]$. If the stepwise execution involves $exec(\mathsf{ext}(t;S);x)$, the states contributed to the trace by this instance of t are exactly the states of $exec(t,x)$. If t occurs within a par construct, the actions of t will be interleaved with other actions. According to Lemma 4.9 and the interleaving principle, the only difference between the execution of t actions in the interleaving $exec(ileave(t;U),x)$ and $exec(t,x)$ is that the state is changed in between t actions.

Suppose that $var(t) \cap var(E) = \{\}$, and that $E[t]$ has no parallel duplication of t. We compare the stepwise executions of $X_1 = Exec(E[t],x)$ and $X_2 = Exec(*\texttt{[}t\texttt{]},x)$, where we are interested in the projection of states on $var(t)$. The stepwise execution is applied to X_1 only, until the first action of t is encountered; i.e., until an evaluation $exec(\mathsf{ext}(t;S),y)$ or $exec(ileave(t;U),y)$ is reached. Since $var(t)$ and $var(E)$ are disjoint, $y\lfloor var(t) = x\lfloor var(t)$. Hence, since t actions only depend on variables in $var(t)$. the next step in X_1 is exactly the same as the next step in X_2, In case of an interleaving, the following step in X_1 may not be a t action, in which case it will not affect the part of the state concerned with $var(t)$. Hence, steps of X_1 are taken until the next t action is encountered. Since there is no parallel duplication of t, this must be identical to the next action of X_2. Since the relevant part of the state has not changed, both actions have the same effect. The same argument applies to all following steps involving t. The only way in which a difference between the executions can be observed is when no instance of t can be reached anymore in X_1, for instance because X_1 is finite.

So far, we have shown that the contributions of t actions to the *execution trees* are identical in X_1 and X_2. That is not quite the same as saying that the contributions to the *traces* are identical, in case of the presence of $\perp$. Suppose a guard in t is evaluated to **false** at a point where there is no other choice of t action available. This causes a $\perp$ in the trace of X_2. But in X_1 it is possible that there is a choice available of an action that is not part of t. Hence, the same **false** guard does not contribute $\perp$ to the trace of X_1. However, in this case the trace of X_1 cannot contain any more contributions by t actions. (The situation for unmatched sync nodes is identical to that for **false** guards.)

From this argument follows the following lemma.

LEMMA 5.1

If $var(t) \cap var(E) = \{\}$, and if $E[t]$ has no parallel duplication of t, then:

For any $p \in Exec(E[t], x)$ there is a $p' \in Exec(*\texttt{[}t\texttt{]}, x)$, such that
$p \lfloor var(t)$ is a prefix of $p' \lfloor var(t)$, or
$p = q +\!\!+ \perp$ and $q \lfloor var(t)$ is a prefix of $p' \lfloor var(t)$.

□

Note that p can end with a $\perp$ state that is not contributed by a t action. Since $\perp$ does not disappear in the projection, we have to make a special provision for $\perp$ in the lemma.

Essentially the same argument can be used to extend the lemma to an environment with two arguments. Compare the stepwise executions of $X_1 = Exec(E[s, t], x)$ and $X_2 = Exec(*\texttt{[}s\texttt{]}\ //\ *\texttt{[}t\texttt{]}, x)$. We assume that $var(E) \cap var(s) = var(E) \cap var(t) = \{\}$, and that there is no parallel duplication of s or t in $E[s, t]$ (meaning that $s\ //\ s$ cannot occur, but $s\ //\ t$ can). In X_2 there is at any step a choice between an s action and a t action. Hence, whenever in X_1 one of these actions is reached, the same action is reached in X_2, with the same relevant part of the state. So certainly any states contributed to the computations by a node from s or t is identical in X_1 and X_2.

The situation with **false** guards (or unmatched sync nodes) is somewhat more complicated. As before, it is possible for a computation of X_2 to end with $\perp$ while the corresponding computation of X_1 can continue with actions that are not part of s or t. But in this case there is another asymmetry between the two executions: Suppose in X_1 a guard from s is reached, and evaluated to **false**. The same then also happens in X_2. In X_1 this leads to deadlock if there is no other choice of action available. But in X_2 there is always at least a choice for an action from t, so that X_2 can only end in $\perp$ if this t action is also a **false** guard. Hence, if X_1 ends in deadlock caused by an s or t action, this does not mean that X_2 ends in deadlock.

LEMMA 5.2

Let $V = var(s) \cup var(t)$. If $V \cap var(E) = \{\}$, and if $E[s, t]$ has no parallel duplication of s nor of t, then:

> For any $p \in \mathit{Exec}(E[s,t],x)$ there is a $p' \in \mathit{Exec}(*\texttt{[}s\texttt{]} \;//\; *\texttt{[}t\texttt{]},x)$, such that
> $p{\downharpoonright}V$ is a prefix of $p'{\downharpoonright}V$, or
> $p = q \mathbin{+\!\!+} \perp$ and $q{\downharpoonright}V$ is a prefix of $p'{\downharpoonright}V$.

□

We now go back to our goal of proving the handshake expansion transformation. This transformation relies on special program sequences called *handshake sequences*.

DEFINITION 5.3 *(Handshake Sequence)*

A handshake sequence is a program of the form

$$t \quad = \quad \texttt{[}e_0\texttt{]};\; s_0;\; \texttt{[}e_1\texttt{]};\; s_1;\; \ldots\; \texttt{[}e_k\texttt{]};\; s_k$$

where each e_j is a guard and each s_j a program, such that

1. $k \geqslant 2$;
2. by the process factorization theorem, $t \equiv t_{even} \;//\; t_{odd}$;
3. $*\texttt{[}t\texttt{]} \equiv *\texttt{[}t_{even}\texttt{]} \;//\; *\texttt{[}t_{odd}\texttt{]}$;
4. $*\texttt{[}t\texttt{]}$ has no deadlock;
5. all variables of t are new variables;
6. for all appropriate environments E, $E[t_{even}, t_{odd}]$ does not have parallel duplication of t_{even} nor of t_{odd}.

□

At the end of this section we present several examples of handshake sequences, but for the proofs we use the general description of t from the definition. Programs satisfying requirement 3 were discussed at the end of Section 4.6 (Example 4.32 and Example 4.33).

Because $t_{even} \;//\; t_{odd} \equiv t$, we have the following lemma.

LEMMA 5.4

Let t be a handshake sequence. Consider a stepwise execution involving t_{even} and t_{odd}.

- t_{even} is finished $\Rightarrow$ t_{odd} has started;
- t_{odd} is finished $\Rightarrow$ t_{even} has started.

Proof:

- Since $k \geqslant 2$, the last action of t_{even} must be or follow $\texttt{[}e_2\texttt{]}$. Since in t this action follows s_1, which is part of t_{odd}, the execution of t_{odd} must have started before t_{even} can finish.
- Since $k \geqslant 2$, t_{odd} contains at least $\texttt{[}e_1\texttt{]}$. Since in t this action follows s_0, which is part of t_{even}, the execution of t_{even} must have started before t_{odd} can finish.

□

Let C^A and C^P be matching sync nodes of the form $\mathsf{sync}_\ell(\mathbf{skip};\mathbf{skip})$. Let F be a two-parameter environment for which ℓ is a new variable, and for which t is a handshake sequence. (It follows that in $F[C^A, C^P]$, C^A can only be matched by C^P, and vice versa.) The transformation consists of two steps, starting with

$$F[C^A, C^P] \;\equiv\; F[(C^A; t_{even}), (C^P; t_{odd})]$$

Proof: Consider the usual X_1 and X_2. All variables in t_{even} and t_{odd} are new variables, so assignments to them are not observable. Hence, the only possible difference between X_1 and X_2 can be the occurrence of deadlock in X_2 caused by t_{even} or t_{odd}. We show that such deadlock cannot occur.

From Lemma 5.2 it follows that, as far as t_{even} and t_{odd} is concerned, X_2 behaves as a prefix of $\texttt{*[}t_{even}\texttt{]}$ // $\texttt{*[}t_{odd}\texttt{]}$. Since t is a handshake sequence, we know that

$$\texttt{*[}t_{even}\texttt{]} \;//\; \texttt{*[}t_{odd}\texttt{]} \equiv \texttt{*[}t\texttt{]} \;\equiv \texttt{*[}t_{even} \;//\; t_{odd}\texttt{]}$$

Now consider only t actions, and compare X_2 with $X_3 = exec(\texttt{*[}t_{even}$ // $t_{odd}\texttt{]}, x)$. In X_2, when t_{even} is reached, it must be following the execution of C^A; according to rule I4, this means that C^P was executed simultaneously, so that t_{odd} has been reached as well. Hence, at this point X_2 involves execution of t_{even} // t_{odd}, just as X_3. Since X_3 has no deadlock, this execution cannot cause deadlock in X_2 either, assuming that no other instances of t_{even} and t_{odd} are reached. (This assumption is correct because otherwise another matched pair of C^A and C^P must have been executed, which would imply parallel duplication of at least one of t_{even} and t_{odd}.) The same holds for all further occurrences of t_{even} and t_{odd}, leading to the conclusion that t actions cannot cause deadlock in X_2.
□

Note that, in the above proof, using $\texttt{*[}t_{even}\texttt{]}$ // $\texttt{*[}t_{odd}\texttt{]}$ for X_3 would not be sufficient, because absence of deadlock in that program might require a sequence of, for instance, one t_{even} and two t_{odd}'s.

The second step of the transformation is

$$F[(C^A; t_{even}), (C^P; t_{odd})] \;\equiv\; F[t_{even}, t_{odd}]$$

Proof: Consider the usual stepwise executions X_1 and X_2. If, in X_1, C^A is reached and a match with C^P is possible, the match is made without observable effect (since the match is **skip**), and both executions continue equivalently. If however C^A is reached in X_1 and no match is yet possible, t_{even} is reached in X_2 but not in X_1. Since the actions of t_{even} itself are not observable, the first observable difference can occur when an action following t_{even} is reached in X_2. But according to Lemma 5.4, if such an action following t_{even} is reached, then t_{odd} must have been reached as well. If t_{odd} is reached in X_2, then C^P must have been reached in X_1, so that a match is now possible, and the earlier situation is reached, where no differences can be observed.

If C^P is reached before C^A, the equivalent argument for t_{odd} holds.

□

This completes the proof of the handshake expansion transformation.

THEOREM 5.5 *(Handshake Expansion)*

If C^A and C^P are matching sync nodes of the form $\mathsf{sync}_\ell(\mathbf{skip};\mathbf{skip})$, and F is a two-parameter environment for which ℓ is a new variable, and for which t is a handshake sequence, then

$$F[C^A, C^P] \equiv F[t_{even}, t_{odd}]$$

□

In t, the first action is one of t_{even}. Therefore, t_{even} is said to initiate the handshake, and is called the ***active*** half of the handshake sequence; similarly, t_{odd} is called ***passive***. This is why we chose the particular superscripts in C^A and C^P. In Section 5.2 we will see that the choice of passive or active has important consequences.

As examples, we present the three handshake sequences that are used most often. Since it is straightforward to check that they do indeed form handshake sequences (i.e., they satisfy requirements 1–4), we omit that proof. E.g., each sequence t can be constructed using state variables insertion (Theorem 4.16) and state variable wait insertion (Lemma 4.22), showing that it is deadlock-free. The initial values follow from the requirements of process factorization.

EXAMPLE 5.6 *(Four-Phase Handshake)*

$$\begin{array}{lllllllll} t & = & l_o\uparrow; & [l_o]; & l_i\uparrow; & [l_i]; & l_o\downarrow; & [\neg l_o]; & l_i\downarrow; & [\neg l_i] \\ t_{even} & = & l_o\uparrow; & & & [l_i]; & l_o\downarrow; & & & [\neg l_i] \\ t_{odd} & = & & [l_o]; & l_i\uparrow; & & & [\neg l_o]; & l_i\downarrow & \end{array}$$

Initial values: $\neg l_o \wedge \neg l_i$. For this handshake sequence, $k = 4$.

□

A *phase* in this context is an assignment followed by a wait. We count the number of phases that occur in sequence; e.g., a four-phase handshake sequence has four phases in sequence.

EXAMPLE 5.7 *(Lazy Four-Phase Handshake)*

$$t \;=\; [\neg l_i];\; l_o\uparrow;\; [l_o];\; l_i\uparrow;\; [l_i];\; l_o\downarrow;\; [\neg l_o];\; l_i\downarrow$$

$$t_{even} \;=\; [\neg l_i];\; l_o\uparrow;\qquad [l_i];\; l_o\downarrow$$

$$t_{odd} \;=\; [l_o];\; l_i\uparrow;\qquad [\neg l_o];\; l_i\downarrow$$

Initial values: $\neg l_o \wedge \neg l_i$. For this handshake sequence, $k = 3$. The reason why it is called 'lazy' will be made clear in Section 5.3.

□

EXAMPLE 5.8 *(Two-Phase Handshake)*

$$t \;=\; l_o := \neg l_i;\;\; [l_o \neq l_i];\;\; l_i := l_o;\;\; [l_i = l_o]$$

$$t_{even} \;=\; l_o := \neg l_i;\qquad [l_i = l_o]$$

$$t_{odd} \;=\; [l_o \neq l_i];\;\; l_i := l_o$$

Initial values: $l_o = l_i$. This handshake sequence has $k = 2$, the minimum possible value.

□

Since the two-phase handshake sequence has fewer actions in sequence, it is potentially more efficient than the four-phase variants. However, it also has a more complex form, which may negate that advantage. Typically, this handshake sequence is only used when the following simplification can be applied.

Suppose that in $F[C^A, C^P]$, the C^A actions can be numbered $C1^A$ and $C2^A$ so that in the execution $C1^A$ and $C2^A$ always alternate, and likewise for C^P. We now apply handshake expansion with the two-phase handshake, which we'll write as

$$F'[C1^A, C1^P, C2^A, C2^P] \;\equiv\; F'[t^1_{even}, t^1_{odd}, t^2_{even}, t^2_{odd}]$$

Because of the alternation between $C1$ and $C2$ actions, we then find that (assuming that initially $\neg l_o \wedge \neg l_i$)

$$pre(t^1_{even}) \Rightarrow \neg l_o \wedge \neg l_i \qquad pre(t^1_{odd}) \Rightarrow \neg l_i$$

$$pre(t^2_{even}) \Rightarrow l_o \wedge l_i \qquad pre(t^2_{odd}) \Rightarrow l_i$$

Using these properties, the handshake can be simplified as

$$\begin{aligned} t^1_{even} &= l_o\uparrow;\qquad\qquad [l_i] \\ t^1_{odd} &= \qquad [l_o];\; l_i\uparrow \end{aligned}$$

$$\begin{aligned} t^2_{even} &= l_o\downarrow;\qquad\qquad [\neg l_i] \\ t^2_{odd} &= \qquad [\neg l_o];\; l_i\downarrow \end{aligned}$$

Note that, using sync duplication (Example 4.26), we can replace a single instance of C by $C; C$, which can then be numbered $C1; C2$. Of course, if we then use the two-phase handshake expansion, the result is identical to using a four-phase handshake expansion for C directly. However, it may be useful to apply this transformation in cases where there is an alternation of three C actions:

$$*[C;\ \ldots\ C;\ \ldots\ C;\ \ldots\ ] \equiv *[C;\ C;\ \ldots\ C;\ \ldots\ C;\ \ldots\ ]$$

The latter program can be written as

$$*[C1;\ C2;\ \ldots\ C1;\ \ldots\ C2;\ \ldots\ ]$$

so that the simplified two-phase handshake expansion can be applied.

If we consider the proof of the Handshake Expansion theorem, we may observe that the important properties of t are that $*[t] = *[t_{even}] \mathbin{//} *[t_{odd}]$, that $*[t]$ has no deadlock, and that Lemma 5.4 holds. It is possible to find programs that satisfy the same requirements, but that are not obtained by process factorization. Handshake expansion can also be performed with such programs. For instance, [24] presents the following alternative two-phase handshake.

$$t_1 = l_o := \neg l_o;\ [l_o = l_i];\ r_o := \neg r_o;\ [r_o = r_i]$$

$$t_2 = l_i := \neg l_i;\ [l_o = l_i];\ r_i := \neg r_i;\ [r_o = r_i]$$

Using the usual stepwise evaluation argument, we can show that

$$*[t_1] \mathbin{//} *[t_2] \equiv *[t]$$

where

$$\begin{aligned} *[t] = {} & (l_o := \neg l_o \mathbin{//} l_i := \neg l_i); \\ & *[(([l_o = l_i];\ r_o := \neg r_o) \mathbin{//} ([l_o = l_i];\ r_i := \neg r_i)); \\ & \quad (([r_o = r_i];\ l_o := \neg l_o) \mathbin{//} ([r_o = r_i];\ l_i := \neg l_i)) \\ &] \end{aligned}$$

(Initial values are $l_o = l_i \wedge r_o = r_i$.) Since $*[t]$ has no deadlock, and Lemma 5.4 can be proven for t_1 and t_2, we have as a variant of the Handshake Expansion theorem that

$$F[C!, C?] = F[t_1, t_2]$$

Note that, based on the form of t, there is no distinction between active and passive halves of the handshake sequence. t_1 and t_2 can be simplified under the same conditions as given for the two-phase handshake expansion. Like the two-phase handshake sequence, t has only four actions in sequence. On the other hand, it uses twice as many variables. Nevertheless, a potential advantage (as explained in the next section) is that both t_1 and t_2 can be considered active.

As a refinement of this alternative two-phase handshake, consider that $t_1 = t_1'; t_1'$ and $t_2 = t_2'; t_2'$ where

$$t_1' \;=\; l_o, r_o := \neg r_o, l_o;\;\; \texttt{[}(l_o \neq r_o \wedge l_o = l_i) \vee (l_o = r_o \wedge r_o = r_i)\texttt{]}$$

$$t_2' \;=\; l_i, r_i := \neg r_i, l_i;\;\; \texttt{[}(l_i \neq r_i \wedge l_o = l_i) \vee (l_i = r_i \wedge r_o = r_i)\texttt{]}$$

if the initial values are $l_o = l_i = r_o = r_i$. The corresponding $*\texttt{[}t\texttt{]}$ is

$$\begin{aligned} *\texttt{[}t'\texttt{]} \;=\; & (l_o, r_o := \neg r_o, l_o \;//\; l_i, r_i := \neg r_i, l_i);\\ & *\texttt{[}\;\; (\texttt{[}(l_o \neq r_o \wedge l_o = l_i) \vee (l_o = r_o \wedge r_o = r_i)\texttt{]};\;\; l_o, r_o := \neg r_o, l_o) \;//\\ & \qquad (\texttt{[}(l_i \neq r_i \wedge l_o = l_i) \vee (l_i = r_i \wedge r_o = r_i)\texttt{]};\;\; l_i, r_i := \neg r_i, l_i)\\ & \texttt{]} \end{aligned}$$

It follows that t_1' and t_2' form a one-phase handshake. t' has only two actions in sequence, an assignment and a wait, which seems the minimum possible for a synchronization. Hence, this is potentially a very fast handshake, at the cost of two extra variables. A similar simplification as for the two-phase handshake can be applied, but this time if sync nodes can be numbered $C1$, $C2$, $C3$, $C4$. As with the alternative two-phase handshake, both halves of the one-phase handshake are active.

5.2 Probes

In the previous section we only considered sync actions where the match is equal to **skip**. Thus, the presented handshake sequences implement only the synchronization behavior of sync actions. If we transform $F[C^A, C^P]$ into $F[t_{even}, t_{odd}]$ when the match of C^A and C^P is $m \neq$ **skip**, we must make sure that the handshake sequence t includes m. Furthermore, if we compare the usual stepwise evaluations X_1 and X_2, it is clear that m must occur after both t_{even} and t_{odd} have been started, and before either one has finished. In terms of a handshake sequence

$$t \;=\; \texttt{[}e_0\texttt{]};\; s_0;\; \texttt{[}e_1\texttt{]};\; s_1;\; \ldots\; \texttt{[}e_{k-1}\texttt{]};\; s_{k-1};\; \texttt{[}e_k\texttt{]};\; s_k$$

$(k \geqslant 2)$, this means that m must occur in one or more of $s_1 \ldots s_{k-1}$. Depending on which effects of m can be observed by the environment, it may be possible to perform m in several steps. In particular, if $C^A = \mathsf{sync}_\ell(f_1; f_2)$ and $C^P = \mathsf{sync}_\ell(g_1; g_2)$ it may be desirable to make f_1 and f_2 part of t_{even} and g_1 and g_2 part of t_{odd}.

A non-**skip** match occurs with communication actions and with probed synchronization actions. In this section we discuss the latter.

Suppose C^A can be probed, but C^P cannot. That is,

$$C^A \;=\; \#C^A\!\uparrow;\;\; \mathsf{sync}_\ell(\textbf{skip};\;\; \#C^A\!\downarrow)$$

$$C^P \;=\; \qquad\quad \mathsf{sync}_\ell(\textbf{skip};\;\; \textbf{skip})$$

In that case, the match is $\#C^A\downarrow$. We can use the above argument, and insert $\#C^A\downarrow$ in one of $s_1, \ldots, s_{k-1}$. For instance, in the four-phase handshake sequence, we can use s_2:

$$\begin{array}{lclll} t'_{even} & = & l_o\uparrow; & [l_i];\ \#C^A\downarrow;\ l_o\downarrow; & [\neg l_i] \\ t_{odd} & = & [l_o];\ l_i\uparrow; & & [\neg l_o];\ l_i\downarrow \end{array}$$

We can then transform C^A into $\#C^A\uparrow; t'_{even}$. However, in the particular case of the probe $\#C^A$, a more efficient implementation exists.

The assignment $\#C^A\uparrow$ must be performed by C^A regardless of whether C^P is reached. If we consider handshake t, then we see that s_0 is performed by t_{even} regardless of whether t_{odd} is reached. Hence, the assignment $\#C^A\uparrow$ can be part of s_0. So, if there is a condition that is set to **true** by s_0 and set to **false** by one of $s_1, \ldots, s_{k-1}$, then this condition can be used instead of the probe $\#C^A$. Regardless of what s_0 is, we know that certainly it makes condition e_1 true, since the factorization conditions require $pre(s_0) \Rightarrow \neg e_1$, and absence of deadlock requires $post(s_0) \Rightarrow e_1$. Furthermore, the synchronization conditions also require that $post(s_k) \Rightarrow \neg e_1$. Hence, e_1 can be used as translation of the probe $\#C^A$, provided that we can show that $post(s_{k-1}) \Rightarrow \neg e_1$ and that e_1 remains false during execution of s_k. Fortunately, this is the case for each of the standard handshake sequences.

Example 5.9

Because $*[t] = *[t_{even}]\ //\ *[t_{odd}]$, there are only two possible forms of t, as discussed at the end of Section 4.6. If k is even (Example 4.32), there is no s_k, and the factorization conditions already require that $post(s_{k-1}) \Rightarrow \neg e_1$, so that there are no additional proof obligations.

Hence, for the four-phase handshake ($k = 4$), l_o implements the probe $\#C^A$.

For the two-phase handshake ($k = 2$), $l_o \neq l_i$ implements the probe $\#C^A$. In cases where the two-phase handshake can be simplified, $l_o \neq l_i$ simplifies likewise.

□

Example 5.10

If k is odd (Example 4.33), we have to prove that e_1 is set to **false** by s_{k-1} rather than by s_k. Indeed, in the lazy four-phase handshake ($k = 3$), e_1 is equivalent to l_o, and s_{k-1} is $l_o\downarrow$, whereas s_k does not involve l_o.

Hence, for the lazy four-phase handshake, just as for the normal four-phase handshake, l_o implements the probe $\#C^A$.

□

As a result, for the standard handshake sequences, we get the probe $\#C^A$ 'for free,' without any additional steps. The same method cannot be used for $\#C^P$, because no part of t_{odd} is executed before t_{even} is reached. Therefore, it is prudent to choose a passive implementation for those synchronization actions that need to be probed. As we have mentioned before, in practice our programs always use at most one of $\#C^A$ and $\#C^P$, not both, so that this choice poses no problems. On the other hand, if we have alternative handshakes where both halves are active, it may be possible to use a similar argument to get both probes for free, so that this choice need not be made at all.

EXAMPLE 5.11

In the alternative two-phase handshake, the first step (corresponding to s_0) has the effect of making $l_o \neq l_i$ **true**, and the second step (s_1) makes it **false** again. Furthermore, s_0 is performed by whichever of t_1 and t_2 is reached first, regardless of whether the other is reached. Hence, $l_o \neq l_i$ implements both probes $\#C!$ and $\#C?$.

□

EXAMPLE 5.12

The one-phase handshake is quite similar to the alternative two-phase handshake, and consequently the negations of the waits implement the probes.

□

To close this section we point out that if $pre(C^P) \Rightarrow \#C^A$, then, after replacement with a normal handshake sequence, $pre(t_{odd}) \Rightarrow e_1$. Since t_{odd} starts with $[e_1]$, this wait is vacuous (Example 4.17), and can be omitted.

EXAMPLE 5.13

The process decomposition transformation (Theorem 4.30) results in a program of the form

$$F[C^A] \ // \ *[[\#C^A \rightarrow t;\ C^P]]$$

The choice of active and passive synchronizations is made because one of the two is probed. Since there is only one C^P action, we choose a four-phase handshake rather than a two-phase handshake, resulting in

$$F[l_o\uparrow;\ [l_i];\ l_o\downarrow;\ [\neg l_i]] \ // \ *[[l_o \rightarrow t;\ l_i\uparrow;\ [\neg l_o];\ l_i\downarrow]]$$

Here we have applied the above optimization of omitting $[l_o]$ when $pre(C^P) \Rightarrow \#C^A$.

□

5.3 Handshake Reshuffling

The handshake expansion theorem can be applied with any handshake sequence

$$t \ = \ [e_0];\ s_0;\ [e_1];\ s_1;\ \ldots\ [e_k];\ s_k$$

with $k \geqslant 2$. Although our semantics does not model time, it nevertheless seems reasonable that longer sequences of actions take more time and are therefore less efficient. This would be a good reason to always use $k = 2$. However, as we have seen, among the standard handshake sequences, the two-phase ($k = 2$) handshake is harder to use than the normal and lazy four-phase handshakes ($k > 2$). In this section, we discuss a transformation, called *handshake reshuffling*, which, in practice, greatly increases the efficiency of handshake sequences with $k > 2$. In addition, handshake reshuffling often helps in distinguishing states of the computation, which is important for the transformations described in Chapter 6.

The proof of the handshake expansion theorem consists of two parts: that there is no deadlock introduced by waits of the handshake, and that the handshake sequence implements the desired synchronization. The synchronization part of this proof relies on Lemma 5.4, which says that one half of the handshake cannot finish before the other half has started. This lemma is true whenever $k \geqslant 2$; in fact, it is true that no action of $s_2, \ldots, s_k$ can be reached unless both halves of the handshake have been started. Therefore, the sequence $s_2; \ldots; [e_k]; s_k$ plays no role in the synchronization part of the handshake expansion theorem. Hence, as far as that part of the proof is concerned, the actions of this sequence may just as well be omitted or at least postponed.

Of course, $s_2; \ldots; [e_k]; s_k$ has its purpose, so it cannot be just omitted. In particular, this part of the sequence is used to prove that $*[t] = *[t_{even}] \; // \; *[t_{odd}]$. This property in turn proves absence of deadlock, through the projection property of Lemma 5.2. This lemma essentially says that $F[t_{even}, t_{odd}]$ behaves as $*[t_{even}] \; // \; *[t_{odd}]$ with respect to actions from t. From the proof of that lemma it is immediately clear that it still holds if there are non-t actions in between the t actions, since that is exactly what happens during an interleaving. Therefore, the projection lemma still holds if t actions are postponed, provided that their relative order is not changed.

DEFINITION 5.14 *(Handshake Reshuffling)*

Given a handshake sequence

$$t \;=\; [e_0];\; s_0;\; [e_1];\; s_1;\; \ldots\; [e_k];\; s_k$$

that, by application of the handshake expansion theorem, occurs as

$$[e_0];\; s_0;\; [e_2];\; s_2;\; [e_4];\; s_4;\; \ldots\; [e_m];\; s_m;\; u_2;\; r_4;\; u_4;\; \ldots\; r_m;\; u_m$$

$$[e_1];\; s_1;\; [e_3];\; s_3;\; [e_5];\; s_5;\; \ldots\; [e_n];\; s_n;\; r_3;\; u_3;\; r_5;\; u_5;\; \ldots\; r_n;\; u_n$$

($m = k$ and $n = k - 1$ if k is even, $m = k - 1$ and $n = k$ if k is odd). If none of the programs r_j and u_j contain an action from t, then the replacement of the above sequences by

$$[e_0];\; s_0;\; [e_2];\; u_2;\; s_2;\; r_4;\; [e_4];\; u_4;\; s_4;\; \ldots\; r_m;\; [e_m];\; u_m;\; s_m$$

$$[e_1];\; s_1;\; r_3;\; [e_3];\; u_3;\; s_3;\; r_5;\; [e_5];\; u_5;\; s_5;\; \ldots\; r_n;\; [e_n];\; u_n;\; s_n$$

is called a reshuffling of the handshake.

□

In words, a reshuffling consists of postponing the actions of $s_2; \ldots ; [e_k] ; s_k$ while maintaining their relative order. The term *reshuffling* has traditionally been used for this transformation, but is actually a misnomer, because the transformation only involves postponing certain parts of the handshake. Specifically, reordering of parts is not allowed. Note also that the above is just a definition; there is no claim that all reshufflings are correct.

EXAMPLE 5.15

For the four-phase handshake, the actions $l_o\downarrow; [\neg l_i]$ of t_{even} and $[\neg l_o] ; l_i\downarrow$ of t_{odd} are candidates for postponement.

For the lazy four-phase handshake, only $l_o\downarrow$ of t_{even}, as well as $[\neg l_o] ; l_i\downarrow$ of t_{odd} can be postponed.

The two-phase handshake cannot be reshuffled, since the last action is $[e_2]$.

□

EXAMPLE 5.16

The alternative two-phase handshake consists of two consecutive one-phase handshakes. Obviously, synchronization is established as soon as the first one-phase handshake has finished, so that the remaining actions are candidates for postponement. Actions $r_o := \neg r_o; [r_o = r_i]$ of t_1 and $r_i := \neg r_i; [r_o = r_i]$ of t_2 can be postponed. With such postponement, the alternative two-phase handshake may be more efficient than the standard two-phase handshake.

The one-phase handshake cannot be reshuffled, since it consists of the minimal sequences that can establish synchronization.

□

Now reconsider the proof of the handshake expansion theorem for a reshuffled handshake sequence. As pointed out, since the sequence $[e_0] ; s_0; [e_1] ; s_1; [e_2]$ has not been changed, the synchronizing behavior of the handshake sequence is unchanged. In the proof of absence of deadlock, we use the projection property that, as far as t actions are concerned, a program containing handshake sequences behaves as a prefix of $*[t_{even}] \;//\; *[t_{odd}]$. This is still true for reshuffled handshakes, provided there is no parallel duplication between actions of t_{even} or t_{odd}. We then point out that

$$*[t_{even}] \;//\; *[t_{odd}] \;\equiv\; *[t] \;\equiv\; *[t_{even} \;//\; t_{odd}]$$

which is also still true (since we have not changed t). Next we consider an execution involving handshakes, looking at t actions only. We conclude that whenever t_{even} or t_{odd} is reached, the stepwise evaluation continues exactly like $*[t_{even} \;//\; t_{odd}]$, which proves absence of deadlock. This last step can no longer be applied, because it is only true for the initial part of the sequence that is not postponed; for any of the postponed actions we must prove separately that they cannot cause deadlock. Since only waits can cause deadlock, the proof requirement for a reshuffling is then to show that none of the postponed waits can cause deadlock. It turns out that this cannot be done for all reshufflings.

Let $F[C^A, C^P]$ be a deadlock-free program. We compare the stepwise evaluations of X_1 involving $F[t_{even}, t_{odd}]$, and X_2 containing reshuffled versions of t_{even} and t_{odd}. Note that X_1 has no deadlock. Both evaluations are identical until the first handshake action in X_1 that has been postponed in X_2 (i.e., at least until $[e_2]$ has been passed). In X_1, this point is followed by execution of the remaining handshake actions. Since these actions are not observable by the environment, and since X_1 has no deadlock, this does not expose any differences. Following s_k in X_1, both evaluations continue identically until the first postponed handshake wait is reached (a postponed handshake assignment is not observable), say $[e_j]$. If e_j is **true**, no difference is exposed, and both evaluations continue till the next postponed wait. But if $\neg e_j$ holds, this is similar to a **false** guard insertion, meaning that there are fewer traces in X_2 than in X_1. Since $[e_j]$ is part of a handshake sequence, $\neg e_j$ holds if and only if s_{j-1} has not been reached (in X_2). We continue the stepwise evaluations; clearly, any step of X_2 can be matched by X_1, until either s_{j-1} is reached in X_2, or until X_2 deadlocks. If s_{j-1} is reached, e_j becomes **true**, and both evaluations continue without exposed differences. Hence, to avoid deadlock, we must make sure that s_{j-1} can be reached.

Suppose, in X_2, $[e_j]$ occurs as $[e_j]; u_j$, and s_{j-1} as $u_{j-1}; s_{j-1}; r_{j+1}$. Then deadlock corresponds to a situation in X_1 where r_{j+1} cannot be reached unless u_j is started. If no such interleaving exists in X_1, then there is no deadlock at $[e_j]$ in X_2 (and vice versa). (r_{j+1} is reached is equivalent to u_{j-1} has finished.) This gives us the following theorem.

THEOREM 5.17 *(Handshake Reshuffling)*

Using the same notation as in the handshake reshuffling definition, let t be a handshake sequence, let

$$t_1 \;=\; t_{even};\; u_2;\; r_4;\; u_4;\; \ldots\; r_m;\; u_m$$

$$t_2 \;=\; t_{odd};\; r_3;\; u_3;\; r_5;\; u_5;\; \ldots\; r_n;\; u_n$$

and let t_1' and t_2' be the corresponding reshuffling.

If $F[t_1, t_2]$ is obtained by handshake expansion, and if, in an execution of $F[t_1, t_2]$, u_{j-1} can always be finished before u_j has started (for all $j > 2$), then

$$F[t_1, t_2] \leqslant F[t_1', t_2']$$

provided that there is no parallel duplication between any actions from t_{even} nor between any actions from t_{odd}.

□

(In this theorem, 'always' means that at any step during the stepwise evaluation there is a choice of interleaving that finishes u_{j-1} before it starts u_j.) Although the theorem accurately describes when a reshuffling is possible, it is not so easy to use in practice. Therefore, we describe some lemmas that are easier to apply and are usually sufficient to prove the correctness of a reshuffling.

To simplify the problem, we consider a situation where we have a number of non-terminating processes $t_1, \ldots, t_n$, executed in parallel:

$$\mathsf{par}(t_1, \ldots, t_n)$$

We assume that for a pair of matching synchronizations C^A and C^P, all instances of C^A occur in one non-terminating process, and all instances of C^P in another. We also assume that all instances of C^A can be ordered before executing the program, and likewise for C^P. Furthermore, we assume that all interaction between processes is through sync actions. Although these assumptions are more strict than our usual ones, they are in fact satisfied by standard CSP programs without shared variables. (Assuming that either C^A corresponds to $C?$ and C^P to $C!$, or vice versa.) Hence, we'll refer to this set of assumptions as the *CSP process model*. In addition to this model, we also assume that there is no deadlock, since if there is already deadlock, there is not much point in proving that reshufflings are deadlock-free.

Note: The restriction that C^A actions can be ordered before execution does not exclude programs of the form

$$[e_1 \rightarrow C^A [] e_2 \rightarrow \mathbf{skip}] ; C^A$$

because in the tree the C^A actions along each path can be ordered. What is excluded is a situation like

$$\mathsf{par}(e_1 \rightarrow C^A, e_2 \rightarrow C^A)$$

where the environment non-deterministically determines the order in which e_1 and e_2 are made **true**. Excluding this situation is hardly restrictive, and helps avoid parallel duplication of handshakes.

DEFINITION 5.18 *(Dependency)*

An action n (a node, guard, or sync), depends on another action m, if, in the execution there is a point where m must be executed before n can finish. n depends on m is written as $m \xrightarrow{*} n$.

n directly depends on m if $m \xrightarrow{*} n$ and there is no other action p such that $m \xrightarrow{*} p \land p \xrightarrow{*} n$; in that case we write $m \rightarrow n$.

□

Clearly, dependencies impose ordering constraints on the execution of actions. Since circular orderings do not exist, the dependencies $m \xrightarrow{*} n$ and $n \xrightarrow{*} m$ together cannot be satisfied. A program that contains this combination of dependencies can never execute either m or n, which usually implies that the program has deadlock. There is one exception: if both dependencies are direct, they can be satisfied by executing m and n simultaneously. This, of course, is only possible with sync actions.

Apart from sync actions, which always depend on each other, dependencies are formed by sequencing within a program (in $\mathsf{ext}(f;g)$, $f \xrightarrow{*} g$), and by a wait and the assignment that makes the wait **true**. (Both these dependencies may, but need not, be direct.) In the

CSP process model we denote an action n in process t_j by $t_j\!:n$. From the assumptions it follows that any dependency $t_i\!:m \xrightarrow{*} t_j\!:n$ with $i \neq j$ must involve a sync dependency. After handshaking expansion, these sync nodes are replaced by handshake sequences, and the dependencies involve the waits of this sequence. In particular, if

$$t \;=\; [e_0];\; s_0;\; [e_1];\; s_1;\; [e_2];\; \ldots\; [e_k];\; s_k$$

is a handshake sequence, then $s_{j-1} \xrightarrow{*} [e_j]$. Since s_{j-1} and $[e_j]$ are in different halves of the handshake sequence, this establishes a dependency between different processes.

The handshake reshuffling theorem says that a reshuffling is valid if there is no dependency $m \xrightarrow{*} n$ where m is a node in u_j and n a node in u_{j-1}. In the program with the reshuffled handshake, this simply means there is no dependency $[e_j] \xrightarrow{*} s_{j-1}$. The converse is then also clear: if $[e_j] \xrightarrow{*} s_{j-1}$ then there is a circular dependency (since always $s_{j-1} \xrightarrow{*} [e_j]$), and the handshake sequence cannot be completed. To verify that the reshuffling does not deadlock at $[e_j]$, we must verify that there is no dependency $[e_j] \xrightarrow{*} s_{j-1}$. Since this is a dependency between two different processes, it must involve either a sync, or an assignment and wait of a handshake sequence. Hence, to find such a dependency, we need only consider sync nodes and handshake assignments that follow $[e_j]$ in the reshuffling. From such nodes we need only consider a dependency to another process.

EXAMPLE 5.19

Let variables l_o and l_i form a four-phase handshake, and r_o and r_i form another four-phase handshake. Variable x is not a handshake variable. Suppose processes p and q contain the following reshuffled sequences.

p: $lo\!\uparrow;\; [li];\; lo\!\downarrow;\; [\neg li];\; x\!\uparrow;\; [ro];\; ri\!\uparrow;\; [\neg ro];\; ri\!\downarrow$

q: $[lo];\; li\!\uparrow;\; [\neg lo];\; ro\!\uparrow;\; [ri];\; li\!\downarrow;\; ro\!\downarrow;\; [\neg ri]$

(Only one action has been postponed, namely $l_i\!\downarrow$.) To check whether $[\neg l_i]$ might cause deadlock, we consider the assignments following it, namely $r_i\!\uparrow$ and $r_i\!\downarrow$; assignment $x\!\uparrow$ need not be considered since it is not part of a handshake sequence. There is no need to consider $r_i\!\uparrow \xrightarrow{*} [\neg r_o]$, since this dependency is within the same process. Hence, we start creating a dependency with $p\!: r_i\!\uparrow \xrightarrow{*} q\!: [r_i]$, which holds according to the handshake sequence. Since $[r_i]$ is in q and precedes $l_i\!\downarrow$, it follows immediately that there is a dependency

$$[\neg l_i] \xrightarrow{*} r_i\!\uparrow \xrightarrow{*} [r_i] \xrightarrow{*} l_i\!\downarrow$$

which is not allowed by the handshake reshuffling theorem. Hence, this is not a valid reshuffling.

□

Most examples are not as simple as the one above. In particular, it might have been that $[r_i]$ was not part of process q, but of some third process. In that case we would repeat the search with assignments following $[r_i]$, until we reach process q. Since there is only a finite number of processes, we know that eventually we must either reach q, or a process must occur twice within the chain of dependencies. In particular, if the chain gets back to p, we can stop with this chain of dependencies and try and find another.

EXAMPLE 5.20

(l_o, l_i) form a four-phase handshake, and (r_o, r_i) and (k_o, k_i) each form a two-phase handshake. Suppose the following sequences occur.

$p1$: $lo\uparrow$; $[li]$; $lo\downarrow$; $[\neg li]$; $ro\uparrow$; $[ri]$

$p2$: $[lo]$; $li\uparrow$; $[\neg lo]$; $ko\uparrow$; $[ki]$; $li\downarrow$

$p3$: $[ro]$; $ri\uparrow$; $[ko]$; $ki\uparrow$

Again, only $l_i\downarrow$ has been postponed, and we check whether $[\neg l_i]$ causes deadlock. As before, we must avoid a dependency $p2$: $l_i\downarrow \xrightarrow{*} p1$: $[\neg l_i]$. Starting at $r_o\uparrow$ we find $p1$: $r_o\uparrow \xrightarrow{*} p3$: $[r_o]$. Since we are interested in chains ending in $p2$, we continue the search with assignments following $p3$: $[r_o]$. The first, $r_i\uparrow$ leads back to $p1$, hence we need not consider it any further. The second, $k_i\uparrow$ has dependency $k_i\uparrow \xrightarrow{*} p2$: $[k_i]$, which precedes $l_i\downarrow$. Hence, we find that

$$[\neg l_i] \xrightarrow{*} r_o\uparrow \xrightarrow{*} [r_o] \xrightarrow{*} k_i\uparrow \xrightarrow{*} [k_i] \xrightarrow{*} l_i\downarrow$$

which shows that the reshuffling is not correct.

□

Using this method, we find the following lemma, which, for the CSP process model, is equivalent to the handshake reshuffling theorem.

LEMMA 5.21

To check the correctness of a reshuffling of a handshake sequence

$t \;=\; [e_0];\; s_0;\; [e_1];\; s_1;\; [e_2];\; \ldots\; [e_k];\; s_k$

do the following. Assume s_{j-1} is part of process q. For p: $[e_j]$ ($j > 2$), construct a chain of dependencies of the form

$$p\colon f_1 \xrightarrow{*} r1\colon [d_1] \xrightarrow{*} r1\colon f_2 \xrightarrow{*} r2\colon [d_2] \xrightarrow{*} \ldots$$

where each f_i is an assignment from a handshake sequence, and each $[d_i]$ is the following wait from the same handshake sequence. (The chain can also contain sync nodes if handshake expansion has not been done completely.) Each chain ends if either process p is reached or process q is reached. If there is no such chain that ends in q before s_{j-1}, then $[e_j]$ does not cause deadlock. The handshake expansion is correct if this is true for all $j > 2$.

□

As the examples illustrate, this lemma is convenient for showing that a reshuffling is wrong. But because all chains following $[e_j]$ must be considered, it is in this form not very practical to prove that a reshuffling is correct. The following two lemmas restrict how many chains need to be considered.

LEMMA 5.22

When using Lemma 5.21 to check a reshuffling of handshake sequence t, no chains starting at assignments from t need to be considered.

Proof: Suppose we are checking for deadlock at $[e_j]$. Regardless of whether reshuffling is done or not, an assignment from t that follows $[e_j]$ can never be reached before $[e_j]$ is passed. If that fact causes deadlock, then the deadlock is independent of the reshuffling and always part of t, which means t is not a handshake sequence.

□

LEMMA 5.23

Suppose, using Lemma 5.21, we are searching for a chain that would prove that p: $[e_j] \xrightarrow{*} q\colon s_{j-1}$. Let $q\colon f$ be an assignment following s_{j-1}, and let p: $[d]$ be a wait following $[e_j]$. If we can show that $f \xrightarrow{*} [d]$, then we need not consider any chains starting beyond $[d]$.

Proof: If a chain starting at $p\colon g$ beyond p: $[d]$ shows that p: $[e_j] \xrightarrow{*} q\colon s_{j-1}$, then there is a cyclic dependency

$$[d] \xrightarrow{*} g \xrightarrow{*} s_{j-1} \xrightarrow{*} f \xrightarrow{*} [d]$$

which does not involve $[e_j]$. Hence, even if there is such a chain, it will be found when the search is done for the other parts of the reshuffled handshakes.

□

Note that, using this lemma, we may not find the guard that actually causes deadlock, only a guard that would cause deadlock if it could be reached.

COROLLARY 5.24

Let [e_j] be part of a handshake expansion replacing synchronization action C. If we are searching for a dependency p: [e_j] $\xrightarrow{*}$ q: s_{j-1}, we need not consider any chains starting after the next occurrence of p: C (or the handshake sequence that replaces it).

Proof: Because of the CSP process model, p: C can only be matched by the corresponding synchronization C' in q. Since instances of C' can be ordered, q: C' follows q: s_{j-1}. Hence, there is a dependency q: C' $\xrightarrow{*}$ p: C, so that the above lemma applies.

□

If there are no more occurrences of C, Corollary 5.24 does not help. But in that unlikely case, the sequence [e_2]; ...; [e_k]; s_k can simply be omitted, because, as explained at the beginning of this section, the very purpose of this sequence is to prevent deadlock and interference when the next instance of C is reached.

The remainder of this section is devoted to some examples of reshufflings.

LEMMA 5.25 *(Lazy Reshuffling)*

Let t be a handshake sequence used to replace synchronization actions C and C'. The lazy reshuffling of the handshake is the reshuffling where [e_k]; s_k is postponed till immediately preceding the next instance of C (or C'), and no other handshake actions are postponed. (This is called lazy because [e_k]; s_k is postponed as far as possible.)

The lazy reshuffling is always correct.

Proof: According to Corollary 5.24, only chains starting between [e_k] and C need to be considered. The only action between them is s_k (if it exists). But since s_k is the last action (or sequence of actions) of the handshake sequence, it does not start any chain.

□

EXAMPLE 5.26

The four-phase handshake ends with [$\neg l_i$] (corresponding to [e_k]), hence the lazy reshuffling, which postpones [$\neg l_i$] as far as possible, is correct.

We already knew this, because the result is exactly the lazy four-phase handshake, except that the latter has [$\neg l_i$] before the first instance of C. Since this first wait is vacuous, there is no difference.

□

EXAMPLE 5.27

We have already pointed out that the alternative two-phase handshake can be reshuffled. In particular, it has a lazy reshuffling. Since t ends with two waits in parallel, both can be postponed. The lazy version of this handshake is

$$t_1 \;=\; [r_o = r_i];\; l_o := \neg l_o;\; [l_o = l_i];\; r_o := \neg r_o$$

$$t_2 \;=\; [r_o = r_i];\; l_i := \neg l_i;\; [l_o = l_i];\; r_i := \neg r_i$$

□

EXAMPLE 5.28 *(Independent Synchronization)*

Let t_1 and t_2 be halves of a handshake sequence. Suppose that t_1 only occurs in the non-terminating process $*[t_1]$. Then any reshuffling of t_2 is correct, because there are no dependencies with $*[t_1]$ other than those of the handshake sequence t itself, and t itself does not have deadlock. In this case, we call t_1 an *independent synchronization.*

When the environment of a program is not (yet) known, in particular for large programs (e.g., a whole chip), it is often assumed that all synchronizations in the environment are independent. It is then the task of the designer of the environment (e.g., the designer of a circuit board) to make sure that the synchronizations are 'independent enough.'

□

Chapter 6

Production Rule Expansion

In this chapter we define the semantics of the PRS language, which is the target of the VLSI design method. The PRS language has no sequential composition; instead, sequencing must be enforced by explicitly waiting for appropriate program states. After having defined the language, we describe under which conditions a HSE program can be transformed into a PRS program. This transformation is called production rule expansion.

6.1 Production Rules

A program in the PRS language is written with *production rules*. A production rule describes part of a gate, and is usually written as

$$e \rightarrow f$$

where e is a guard and f an assignment. (Production rules derive their name from their notational similarity with the productions of a grammar.) Usually, f is a constant assignment to a boolean variable, i.e., $u\uparrow$ or $u\downarrow$. A program written with production rules is called a *production rule set*, and is indeed described by a set of production rules. Parallel composition of such programs is equivalent to the use of set union. Performing the assignment f is called the *firing* of the production rule. A production rule can only fire when its guard is true. The execution of a production rule set is usually described as the continuous concurrent firing of all production rules with true guards. In order to describe transformations involving production rules, we must first express the meaning of production rules within the framework of our semantics. There are several ways in which this can be done.

The first method is called the *gate model* of production rules. A gate is a program of the form

$$*[[\ e_1 \longrightarrow f_1 \ [] \ e_2 \longrightarrow f_2 \ [] \ \ldots \ [] \ e_k \longrightarrow f_k \ ]]$$

such that $\langle \bigcap j : 0 < j \leqslant k : \mathit{changes}(f_j) \rangle \neq \{\}$. If t is such a gate, we require that $\mathit{changes}(\mathcal{V}(t)) \cap \mathit{changes}(t) = \{\}$. In words, there is at least one variable to which all assignments in a gate assign, and the environment cannot change any variable that is changed by a gate. Normally, a gate changes exactly one boolean variable, and has the form

$*[[\ e_u \longrightarrow u\uparrow\ [\!]\ e_d \longrightarrow u\downarrow\ ]]$

This would be the gate of variable u; the environment can use u, but is not allowed to change it. Here, e_u is called the *up-guard* or *pull-up* of u; e_d is called the *down-guard* or *pull-down* of u. The gate can have more than one pull-up or pull-down, but this can always be rewritten: if $e1_u$ and $e2_u$ are pull-ups, they can be replaced by the single pull-up $e1_u \vee e2_u$.

EXAMPLE 6.1

A gate of the form

$*[[\ a \wedge b \longrightarrow u\downarrow\ [\!]\ \neg a \vee \neg b \longrightarrow u\uparrow\ ]]$

is called a nand gate (with inputs a and b, and output u). It has the special property that $e_u = \neg e_d$; gates with this property are called *combinational.*

A gate of the form

$*[[\ a \wedge b \longrightarrow u\downarrow\ [\!]\ \neg a \wedge \neg b \longrightarrow u\uparrow\ ]]$

is called a Muller C element (or simply 'C element'). It is not combinational: there are states where $\neg e_u \wedge \neg e_d$ holds. Gates with this property are called *state-holding.*

□

In the gate model, a production rule set has the form

$$\mathsf{par}(t_1, \ldots, t_n)$$

where each t_j is a gate. Since each gate is part of the environment of the other gates, no two gates can assign to the same variable. Hence, $t_1, \ldots, t_n$ forms indeed a set, even though the par operator allows for a bag. Note that a production rule set is a non-terminating process.

Because production rules are intended to be mapped directly to a VLSI technology such as CMOS, and also to make programming with production rules easier, some restrictions are imposed on the form of production rules. However, we start with a definition that applies to all programs, not just to production rule sets.

DEFINITION 6.2 *(Stability)*

A guard e is called stable, if, whenever the stepwise execution reaches a point where e is a possible next action and e holds, then e stays true until the execution has passed the guard.

□

Typically, all waits in a program are stable. It is hard to deal with an unstable wait `[e]`, because an execution may never pass the guard, even if the environment makes it occasionally true. However, unstable guards are occasionally used in choices. Because passing a guard has no effect on the state, and because an environment can only observe the state of a program, in practice the environment is prevented from falsifying a stable guard until the completion of the first observable assignment following the guard.

EXAMPLE 6.3

The guards from a (factorized) handshake sequence are stable. For instance, in the four-phase handshake sequence

$$t_{even} \;=\; l_o\uparrow; \qquad [l_i];\; l_o\downarrow; \qquad [\neg l_i]$$

$$t_{odd} \;=\; \qquad [l_o];\; l_i\uparrow; \qquad [\neg l_o];\; l_i\downarrow$$

we know that, if $[\neg l_o]$ evaluates to **true**, it will stay **true** until after execution of $l_i\downarrow$, which is the first assignment following the wait. If the handshake sequence is reshuffled, other assignments may occur between $[\neg l_o]$ and $l_i\downarrow$, so that in that case an even stronger property than stability holds.

□

In a production rule $e \to f$, there is only one assignment that follows guard e, namely f. Suppose we want e to be stable. Then, if at some point of the execution e is reachable and holds, the environment may not falsify e until f has been executed. But following that execution e is immediately reachable again, which would again prevent the environment from falsifying it. Hence, the only way in which e can become false is if it is falsified by f itself. A production rule $e \to f$ where f falsifies e is called *self-invalidating*. Because it is undesirable to restrict production rules to be self-invalidating, we have a slightly weaker definition of stability for production rules.

DEFINITION 6.4 *(Stability of Production Rules)*

A production rule $e \to f$ is stable if e can only be changed from **true** to **false** by f or in a state x where $f(x) = x$.

□

Normally, we require that all guards in a production rule set are stable under this definition (unstable guards are discussed in Section 6.2). In fact, the VLSI implementation may require this in order to work correctly. If $f(x) = x$, the assignment (i.e., the firing) is vacuous (Example 4.11); a firing that is not vacuous is called *effective*.

The second method for defining the meaning of production rules is called the *single selection model* of production rules. Here a production rule set with rules

$$e_1 \to f_1 \quad \ldots \quad e_n \to f_n$$

is modeled by the program

$$*[[\; e_1 \longrightarrow f_1 \;[]\; \ldots \;[]\; e_n \longrightarrow f_n \;]]$$

Again, if t is a program of this form, we require that $changes(\mathcal{V}(t)) \cap changes(t) = \{\}$, i.e., that the environment does not change any variable that is changed by the production rule set. Unlike the gate model, there is no restriction on the relation between f_i and f_j, other than that the rules must form a set (by definition of the tree which this program represents). The obvious difference between the gate model and the single selection model is that in the latter there is no concurrency between production rules. This makes it easier to work with the single selection model, but makes it also (seemingly) less realistic, since most VLSI implementations do in fact exhibit such concurrency. However, the following theorem explains the usefulness of this model.

THEOREM 6.5

If all guards in a production rule set are stable, and all assignments are single constant assignments to booleans (i.e., of the form $u\uparrow$ or $u\downarrow$), then the gate model and the single selection model are equivalent.

Proof: Consider a production rule set with rules $e_j \rightarrow f_j$. Let t_1 be the single selection model implementation of that set, and let t_2 be the same set expressed in the gate model. Let X_1 and X_2 be the respective stepwise evaluations.

Clearly, $t_1 \geqslant t_2$, because any step taken by t_1 can also be taken by t_2 (i.e., the production rules in the gate model can be executed in sequence, instead of interleaved). Hence, we must prove that any step taken by t_2 can also be taken by t_1. Suppose both evaluations are in state x, X_1 has reached (but not made) the choice between all production rules, and X_2 next executes f_j. We distinguish three cases.

- $e_j(x)$. Then in X_1 the guard e_j can be selected, followed by the execution of f_j. No differences are observed.
- $\neg e_j(x) \wedge f_j(x) = x$. Simply do nothing in X_1, since the effect of f_j in X_2 is not observable.
- $\neg e_j(x) \wedge f_j(x) \neq x$. Since f_j is reached in X_2, e_j must have been true in some earlier state, and must have been falsified after the guard in X_2 was passed. Because e_j is stable, it can only have been falsified in a state x' where $f_j(x') = x'$. Since now $f(x) \neq x$, an assignment g must have been executed that changed the state from x_1 to x_2 such that

$$f_j(x_1) = x_1 \quad \wedge \quad g(x_1) = x_2 \quad \wedge \quad f_j(x_2) \neq x_2$$

Say $f_j = u\uparrow$. Then apparently $x_1(u) \wedge \neg x_2(u)$, so that the only possible g is $u\downarrow$. But then $\mathit{changes}(f_j) \cap \mathit{changes}(g) = \{u\}$, so that f and g are part of the same gate, which means that g cannot be reached while the execution has passed e_j but not f_j. Hence, this case cannot occur.

□

For arbitrary f_j and g satisfying the relationships with x_1 and x_2 mentioned in the proof, it can be shown that $\mathit{changes}(g) \subseteq \mathit{changes}(f_j) \cup \mathit{depends}(f_j)$. Hence, the theorem holds as long as $\mathit{depends}(f_j) \subseteq \mathit{changes}(f_j)$. In particular, in the theorem we assumed that f_j

was a constant assignment, so that *depends*$(f_j) = \{\}$ (Example 2.12). From now on we will always assume that f in a production rule $e \rightarrow f$ is a single constant assignment to a boolean variable.

From the theorem it follows that we can write a program in the form of the single selection model, then, assuming all guards are stable, convert it to a program in the form of the gate model, and map that program to a VLSI implementation.

DEFINITION 6.6 *(Interference of Production Rules)*

Two production rules $e_u \rightarrow u\uparrow$ and $e_d \rightarrow u\downarrow$ are interfering if during the program's execution a state x occurs such that $e_u(x) \wedge e_d(x)$.

□

Note that this definition differs from the normal definition of interference, Definition 4.7. Since conflicting assignments are always part of the same gate, no interference according to the standard definition can occur. We usually require that production rules are non-interfering, since this is required by most VLSI implementations. Also, there is hardly ever a use for interfering production rules in a program.

Considering the difference between the single selection model and the gate model, it may appear that a more general, and maybe more obvious, model is one where each production rule $e \rightarrow f$ is modeled by

$$*[[e \rightarrow f]]$$

and a production rule set corresponds to the parallel composition of such programs. However, this is in fact a much more restrictive definition of production rules, because of the following. Consider a firing of a production rule $e_u \rightarrow u\uparrow$. Suppose that after the firing e_u still holds, the guard is passed again, but $u\uparrow$ is not executed. Since in this state u holds, e_u may be falsified and eventually e_d can become **true**. At that point, both $u\downarrow$ and $u\uparrow$ are reachable, causing interference, even though the non-interference requirement for production rules has not been violated. With this model there is only one possible way to avoid such interference, namely by defining each production rule $e \rightarrow f$ so that f falsifies the guard e. As mentioned before, restricting ourselves to such self-invalidating production rules is undesirable. Therefore, this is not a useful way to define production rules.

We present one more method of defining production rules, called the *atomic assignment model* of production rules. This definition has concurrency between all production rules, and does not suffer from the interference problem described above. Here, rules $e_u \rightarrow u\uparrow$ and $e_d \rightarrow u\downarrow$ are modeled by

$$u := e_u \vee u \quad // \quad u := \neg e_d \wedge u$$

Hence, each production rule corresponds to a single assignment, and a production rule set is the parallel composition of these assignments. The above problem does not occur, because there are no actions between the evaluation of e_u and the assignment to u (single assignments are always atomic in our semantics). If $\neg e_u$, the first assignment reduces to the vacuous $u := u$, and likewise for the second assignment when $\neg e_d$. If at some point $e_u \wedge e_d$, then the first assignment corresponds to $u\uparrow$ and the second to $u\downarrow$. Hence, in this model, interference of production rules is identical to the standard definition of interference. Stability, on the other hand, is not clearly expressed in this model. If production rules are stable and non-interfering, the gate model and the atomic assignment model are equivalent (and hence equivalent to the single selection model). We tend to use the single selection and gate models.

In addition to stability and non-interference, a particular technology often imposes additional constraints on the form of production rules. For instance, the standard CMOS implementation of production rules [?] allows only pull-ups where, when written in disjunctive normal form, all literals occur in negated form. Hence, $\neg a \wedge \neg b \rightarrow u\uparrow$ can be directly implemented, but $a \wedge b \rightarrow u\uparrow$ cannot. Likewise, pull-downs can only have positive literals.

Another example is the length of feedback chains. Say guard e_u of $e_u \rightarrow u\uparrow$ becomes true. Because of the stability requirement, e_u can only be changed when $u\uparrow$ is reached. This implies that there is some chain of variable transitions, started by $u\uparrow$, that eventually falsifies e_u. A technology may impose a minimum length (in terms of number of variable transitions) on the length of such a feedback chain. (This minimum length is determined by the gain as well as the delay of circuit components.) In a self-invalidating gate, the transition $u\uparrow$ itself invalidates e_u, giving a chain of length one. In, for instance, the standard CMOS implementation, feedback chains of length one are not allowed. Because of the earlier requirement about the use of negated literals, a feedback chain in CMOS must have odd length; it turns out that length 3 is sufficient for CMOS. Hence, for the standard CMOS implementation, to meet the requirements on the minimum length of feedback chains, it is sufficient to exclude self-invalidating production rules. This is often done by strengthening the definition of stability (Definition 6.4), by only allowing e to become **false** when $f(x) = x$.

EXAMPLE 6.7

The single-gate program

$*[[\ \neg u \longrightarrow u\uparrow\ [\!]\ u \longrightarrow u\downarrow\ ]]$

cannot be implemented directly in CMOS, because it contains self-invalidating production rules. In terms of the program semantics, this program corresponds to $*[u\uparrow; u\downarrow]$. Note that this gate can also be written as

$*[[\ \mathbf{true} \longrightarrow u\uparrow\ [\!]\ \mathbf{true} \longrightarrow u\downarrow\ ]]$

which does not have self-invalidating production rules, but instead violates the non-interference requirement.

□

If a production rule set with self-invalidating production rules is generated, it can be implemented by first inserting extra state variable transitions in the feedback chains. However, it is usually better to insert these state variables at an earlier stage of program development, when still working with sequential program sequences, so that the same state variables may be of use when generating the other production rules. The use of state variables for generating production rules is explained in Section 6.3.

6.2 Straight-Line Programs

It is easiest to implement a program with production rules if the program does not have any choices. Such a program is called a straight-line program.

DEFINITION 6.8 *(Straight-Line Program)*

A straight-line program is a program without any choices (i.e., every node in the tree has at most one child).

□

Note that a straight-line program can include par constructs. In order to implement a program with production rules, the program is first written as the parallel composition of a number of straight-line programs. Normally, each straight-line program is a non-terminating process.

For the implementation with production rules, we require that all waits in a straight-line program are stable, as defined in Definition 6.2. This enables two simple transformations.

LEMMA 6.9

If $[e_1]$ and $[e_2]$ are stable, then

$$[e_1]; [e_2] \quad \equiv \quad [e_1 \wedge e_2]$$

Proof: If $[e_1]$ is passed, stability requires that e_1 remains **true** until the first assignment that follows it. Since that assignment also follows $[e_2]$, we conclude that when $[e_2]$ is passed, $e_1 \wedge e_2$ holds. Hence, $[e_1]; [e_2]$ is equivalent to $[e_1]; [e_1 \wedge e_2]$. It is then straightforward to see that the first $[e_1]$ can be omitted.

□

LEMMA 6.10

If $[e_1]$ and $[e_2]$ are stable, then

$$[e_1] \;//\; [e_2] \quad \equiv \quad [e_1 \wedge e_2]$$

Proof: The parallel interleaving leads to the two sequences $[e_1]; [e_2]$ and $[e_2]; [e_1]$. According to the previous lemma each is equivalent to $[e_1 \wedge e_2]$.

□

Next we discuss a transformation that replaces choices by the parallel composition of straight-line programs. Consider a program (a non-terminating process) of the form

```
*[[ $e_1 \longrightarrow$ $t_1$
  [] $e_2 \longrightarrow$ $t_2$
 ]]
```

Assume that for every node, guard, and sync n_1 in t_1, and for every node, guard, and sync n_2 in t_2,

$$pre(n_1) \Rightarrow \neg e_2 \quad \wedge \quad pre(n_2) \Rightarrow \neg e_1$$

Recall that $post(e_1)$ is the subset of $pre(e_1)$ consisting of all states where e_1 holds. Therefore

$$x \in pre(e_1) \wedge x \notin post(e_1) \Rightarrow \neg e_1$$

Furthermore, $post(e_1) \subseteq pre(t_1)$, so that, by the above assumption, $post(e_1) \Rightarrow \neg e_2$. Hence, from the assumption, it follows that

$$pre(e_1) \Rightarrow \neg e_1 \vee \neg e_2$$

(and, of course, $pre(e_1) = pre(e_2)$). In words, $\neg e_1 \vee \neg e_2$ means that e_1 and e_2 are mutually exclusive (at least, when they are reached by the execution).

Under the assumption, we can replace the above program by

$$*[[e_1];t_1] \;//\; *[[e_2];t_2]$$

(this transformation is called *choice decomposition*).

Proof: Consider the usual stepwise evaluations, X_1 and X_2. When the choice in X_1 is first reached, the interleaving in X_2 results in a choice between the same two guards. Suppose e_1 holds (and, therefore, $\neg e_2$). Both evaluations reach t_1. The difference is that, in X_2, at every step there is a choice guarded by e_2. As long as t_1 is not finished, $\neg e_2$ holds, and therefore this choice does not expose any differences (by the **false** guard insertion lemma). Once t_1 is finished, we are back in the original state, where both evaluations are at the choice between e_1 and e_2.
□

We summarize this result in a theorem.
THEOREM 6.11 *(Choice Decomposition)*

If, for every node, guard, and sync n_1 in t_1, and for every node, guard, and sync n_2 in t_2,

$$pre(n_1) \Rightarrow \neg e_2 \quad \wedge \quad pre(n_2) \Rightarrow \neg e_1$$

then

```
*[[ $e_1 \longrightarrow$ $t_1$          *[[$e_1$]; $t_1$ ]
  [] $e_2 \longrightarrow$ $t_2$     ≡    //
 ]]                       *[[$e_2$]; $t_2$ ]
```

□

The condition that $pre(n_1) \Rightarrow \neg e_2$, etc., is not always satisfied. If it is not, we can apply the following transformation.

$$
\begin{array}{ll}
 & *[[e_1 \rightarrow t_1 \,[]\, e_2 \rightarrow t_2]] \\
\equiv & \{\text{state variable insertion}\} \\
 & *[[e_1 \rightarrow u\uparrow; t_1; u\downarrow \,[]\, e_2 \rightarrow v\uparrow; t_2; v\downarrow]] \\
\equiv & \{\text{guard strengthening, Lemma 4.19}\} \\
 & *[[e_1 \wedge \neg v \rightarrow u\uparrow; t_1; u\downarrow \,[]\, e_2 \wedge \neg u \rightarrow v\uparrow; t_2; v\downarrow]]
\end{array}
$$

Here, for any node, guard, or sync n_1 in $t_1; u\downarrow$, we have $pre(n_1) \Rightarrow u \Rightarrow \neg(e_2 \wedge \neg u)$. If we assume that e_1 is stable, then $pre(u\uparrow) \Rightarrow e_1$. If we also assume that $pre(e_1) \Rightarrow \neg e_1 \vee \neg e_2$, then we can conclude that $pre(u\uparrow) \Rightarrow \neg e_2$. Hence, for any n_1 in $u\uparrow; t_1; u\downarrow$, $pre(n_1) \Rightarrow \neg(e_2 \wedge \neg u)$. Using the same argument for the other alternative, this establishes the conditions necessary to apply the previous theorem. We combine these transformations in a single theorem.

THEOREM 6.12 *(Choice Decomposition)*

If e_1 and e_2 are stable, and if $pre(e_1) \Rightarrow \neg e_1 \vee \neg e_2$, then

$$
\begin{array}{l}
*[[\; e_1 \longrightarrow t_1 \\
\;\;[]\; e_2 \longrightarrow t_2 \\
\;\;]]
\end{array}
\equiv
\begin{array}{l}
*[[e_1 \wedge \neg v];\; u\uparrow;\; t_1;\; u\downarrow\;] \\
// \\
*[[e_2 \wedge \neg u];\; v\uparrow;\; t_2;\; v\downarrow\;]
\end{array}
$$

where u and v are new variables.

□

It is not always possible to guarantee that guards are mutually exclusive (i.e., that $pre(e_1) \Rightarrow \neg e_1 \vee \neg e_2$). For instance, a selection between two channels (using probes) coming from independent processes, as in a channel merge, can find both probes **true** simultaneously. Define

$$
arbiter(e_1, e_2) \equiv *[[\;
\begin{array}{lll}
e_1 \wedge \neg v & \longrightarrow & u\uparrow \\
[]\; e_2 \wedge \neg u & \longrightarrow & v\uparrow \\
[]\; \neg e_1 & \longrightarrow & u\downarrow \\
[]\; \neg e_2 & \longrightarrow & v\downarrow \\
]]
\end{array}
$$

If u and v are only assigned by this program, and initially $\neg u \wedge \neg v$, then always $\neg u \vee \neg v$. Furthermore, assuming e_1 and e_2 are stable, it is straightforward to show that

$$
\begin{array}{l}
*[[\; e_1 \longrightarrow t_1 \\
\;\;[]\; e_2 \longrightarrow t_2 \\
\;\;]]
\end{array}
\equiv \; arbiter(e_1, e_2) \; // \;
\begin{array}{l}
*[[\; u \longrightarrow t_1 \\
\;\;[]\; v \longrightarrow t_2 \\
\;\;]]
\end{array}
$$

where the last program has mutually exclusive guards. Of course, this has merely shifted the problem. However, this is acceptable if the arbiter can be implemented as a single circuit element (with two outputs). [14] presents a CMOS implementation of such an element. (A circuit element is usually called a gate, but the arbiter does not satisfy our definition of gates.)

A similar solution exists for the case where guards are mutually exclusive but not stable. This happens normally only when a guard contains a negated probe, $\neg\#C$. Since $\#C$ can become **true** at any moment, this guard is unstable (although $\#C$ is stable). We present the following transformations without proof. Let U and V be synchronization actions.

```
   *[[ $e \longrightarrow\ U$ ▯ $\neg e \longrightarrow\ V$ ]]
 ≡ {use active half of lazy four-phase handshake}
   *[[ $e\ \longrightarrow$  [$\neg u_i$]; $u_o\uparrow$; [$u_i$]; $u_o\downarrow$
    ▯ $\neg e \longrightarrow$  [$\neg v_i$]; $v_o\uparrow$; [$v_i$]; $v_o\downarrow$
   ]]
 ≡
   *[[ $e \wedge \neg v_o \longrightarrow$  [$\neg u_i$]; $u_o\uparrow$
    ▯ $\neg e \wedge \neg u_o \longrightarrow$  [$\neg v_i$]; $v_o\uparrow$
    ▯ $u_i \longrightarrow\ u_o\downarrow$
    ▯ $v_i \longrightarrow\ v_o\downarrow$
   ]]
 ≡
   *[[ $e \wedge \neg v_o \wedge \neg u_i \longrightarrow\ u_o\uparrow$
    ▯ $\neg e \wedge \neg u_o \wedge \neg v_i \longrightarrow\ v_o\uparrow$
    ▯ $u_i \longrightarrow\ u_o\downarrow$
    ▯ $v_i \longrightarrow\ v_o\downarrow$
   ]]
```

This last program is called *synchronizer*(e). With the synchronizer, we can replace $*$[[$e \rightarrow t_1$▯$\neg e \rightarrow t_2$]] by

```
   *[[ $\#U \longrightarrow\ U$; $t_1$ ▯ $\#V \longrightarrow\ V$; $t_2$ ]]
 ≡ {other half of lazy four-phase handshake}
   *[[ $u_o \longrightarrow\ u_i\uparrow$; [$\neg u_o$]; $u_i\downarrow$; $t_1$
    ▯ $v_o \longrightarrow\ v_i\uparrow$; [$\neg v_i$]; $v_i\downarrow$; $t_2$
   ]]
 ≡ {reshuffling; other half is independent handshake}
   *[[ $u_o \longrightarrow\ u_i\uparrow$; $t_1$; [$\neg u_o$]; $u_i\downarrow$
    ▯ $v_o \longrightarrow\ v_i\uparrow$; $t_2$; [$\neg v_i$]; $v_i\downarrow$
   ]]
```

Note that $\#U$ and $\#V$ are stable (positive probes always are), and, because of the way the synchronizer is written, mutually exclusive. From the second program it follows that

u_i and v_i are mutually exclusive, which means they can be replaced by a single variable z. Renaming u_o to u and v_o to v, this gives us

```
synchronizer(e) ≡ *[[ e ∧ ¬v ∧ ¬z  ⟶ u↑
                   ▯ ¬e ∧ ¬u ∧ ¬z ⟶ v↑
                   ▯ z ⟶ u↓
                   ▯ z ⟶ v↓
                  ]]
```

(initially $\neg z$). Hence

```
*[[ e  ⟶ t1                              *[[ u ⟶ z↑; t1; [¬u]; z↓
  ▯ ¬e ⟶ t2    ≡   synchronizer(e) //     ▯ v ⟶ z↑; t2; [¬v]; z↓
 ]]                                       ]]
```

This last program in fact satisfies the conditions of Theorem 6.11. As with the arbiter, this solution is acceptable if the synchronizer can be implemented as a single element (the synchronizer is rather similar to the arbiter).

6.3 Implementation with Production Rules

Before a program is implemented with production rules, it is first transformed into a parallel composition of straight-line programs. This is done with process decomposition (Section 4.5) and the transformations presented in the previous section. We also assume that all synchronizations have been removed (using handshake sequences), that all variables are booleans and all assignments constant assignments, and that all guards are stable (with the exception of the *arbiter* and *synchronizer* elements discussed in the previous section). In this section we describe the implementation of a straight-line program with a production rule set. The parallel composition of straight-line programs is then just the parallel composition (union) of such production rule sets.

Recall that a vacuous firing of a production rule $e \rightarrow f$ can occur in states x where $e(x) \wedge f(x) = x$. Since a vacuous firing does not change the state, it can be repeated any number of times, as long as the environment does not change the state. Normally, such firings do not contribute to the trace, because of the stuttering axiom. However, if an infinite number of vacuous firings occurs in sequence, without state changes, then the trace ends with the special symbol ∞. Say we want to implement straight-line program t_1 with production rule set t_2. Normally, t_1 does not generate traces ending in ∞, which would mean that t_2 is not allowed to contain any vacuous firings. As pointed out before, this requires that all production rules are self-invalidating, which is undesirable. Because of the way production rules are mapped to circuits, it is acceptable if there can be vacuous firings, as long as there is always at least a state change possible. An execution where a state change is possible, but does not occur because at every step a vacuous firing is chosen, is called *unfair*. Fairness is a complex issue [7], and is mostly beyond the scope of this text. To enable us to deal with the issue of vacuous firings, we define a function *fair* that removes unfair traces from a set of traces.

$$\mathit{fair} : \textbf{set of}\,(\mathit{Trace}) \rightarrow \textbf{set of}\,(\mathit{Trace})$$

An unfair trace, for our purposes, has the form $p \mathbin{+\!\!+} \infty$. But it is only unfair if, after generating trace p, another state change was possible, i.e., if there is a q such that $p \mathbin{+\!\!+} q$ is in the trace set.

DEFINITION 6.13 *(Fair Traces)*

Function $\mathit{fair} : \textbf{set of}\,(\mathit{Trace}) \rightarrow \textbf{set of}\,(\mathit{Trace})$ is defined as

$$\begin{aligned} \mathit{fair}(R) = \; & \left\langle \bigcup p \; : \; p \in R \wedge \langle \forall q \; : : \; p \neq q \mathbin{+\!\!+} \infty \rangle \; : \; \{p\} \right\rangle \\ & \cup \left\langle \bigcup p \; : \; p \mathbin{+\!\!+} \infty \in R \wedge \langle \forall q \; : : \; p \mathbin{+\!\!+} q \notin R \vee p \mathbin{+\!\!+} q = p \mathbin{+\!\!+} \infty \rangle \; : \; \{p \mathbin{+\!\!+} \infty\} \right\rangle \end{aligned}$$

□

Using *fair* we define a modified version of the implementation relation, which only requires that fair traces cannot be distinguished. Except for the addition of *fair*, this definition is identical to Definition 3.21.

DEFINITION 6.14 *(Fair Implementation)*

t is implemented by s under fair execution means

$$\begin{aligned} t \leqslant s \;\equiv\; & \mathcal{V}(t) \subseteq \mathcal{V}(s) \\ \wedge\; & E \in \mathcal{V}(t) \;\Rightarrow\; \mathcal{I}(E[t]) \subseteq \mathcal{I}(E[s]) \\ \wedge\; & \langle \forall E \; : \; E \in \mathcal{V}(t) \; : \\ & \quad \langle \forall x \; : \; x \in \mathcal{I}(E[t]) \; : \\ & \quad \mathit{fair}(\mathit{Exec}(E[s], x)\lfloor \mathit{var}(E)) \supseteq \mathit{fair}(\mathit{Exec}(E[s], x)\lfloor \mathit{var}(E)) \\ & \rangle \quad \rangle \end{aligned}$$

□

The corresponding equivalence is the fair equivalence, $\equiv_f$. The above definition also excludes executions where the state does change, but this state change is not observable by the environment (this is not essential for the problem with the vacuous firings).

Note that *fair* is introduced mainly to deal with production rules. In particular, *fair* is intended to remove unfair interleavings, not to make arbitrary choices fair. For instance, consider

```
[true ⟶  u↑
[]true ⟶  *[skip]
]
```

If the initial state is σ, and $\sigma[u\uparrow] = \tau$ such that $\sigma \neq \tau$, then this program generates two traces, $\sigma +\!\!+ \tau$ and $\sigma +\!\!+ \infty$. According to the definition of *fair*, the latter trace is unfair. However, since only a single choice is made, fairness is not an issue for this program, and both traces are equally valid. This problem is avoided if every non-deterministic choice is either a result of interleaving, or each alternative of the choice can be identified by a different state change. (E.g., the above example does not cause problems if the second alternative is changed to $v := \neg v; *[\mathbf{skip}]$.)

When we implement program t_1 with production rule set t_2, we usually want $t_1 \equiv_f t_2$ rather than $t_1 \leqslant_f t_2$, because if equivalence does not hold, that usually means that the number of interleavings has been reduced. Since interleavings often affect the efficiency of a program, and since most efficiency-based design decisions are made before production rules are generated, we do not want to change the interleavings.

EXAMPLE 6.15

Assume l_o and l_i form a four-phase handshake sequence, and consider

$lo\uparrow;\ [li];\ lo\downarrow;\ [\neg li];\ t;\ \ldots$

If the other half of the handshake is independent, the following reshuffling is possible.

$lo\uparrow;\ [li];\ lo\downarrow;\ t;\ [\neg li];\ \ldots$

It may well be that this reshuffling is more efficient than the original, because it allows actions from t before the environment performs $l_i\downarrow$. Therefore, we do not want an implementation with production rules to be equivalent to the first program, because that would remove interleavings where t happens before $l_i\downarrow$.

To avoid such unwanted implementations, we tend to use as little information about the environment as possible when generating production rules. In particular, we assume that the environment obeys the handshake protocol, but do not assume anything about how the environment interleaves handshake sequences, other than that the given straight-line program does not deadlock

□

Next, we consider the implementation of a program t_1 with production rule set t_2, such that $t_1 \equiv_f t_2$. We describe the necessary conditions in two parts, safety and progress. Let t_1 be a straight-line program. We assume that there are no sync nodes, that all guards are stable, and that all assignments are constant assignments. Assume that if $\mathsf{par}(s_1, s_2, \ldots)$ occurs in t_1, then no assignment f occurs in both s_1 and s_2. We also assume that t_1 is a non-terminating process that does not cause deadlock and does not create traces ending in ∞. Let t_2 be a production rule set with production rules $e_j \rightarrow f$. For convenience, we assume stability of the production rules, although that is not strictly necessary for this part of the proof. Since we assume stability, we will use the single selection model for t_2. Let n be a guard or node of t_1, $x \in \mathit{pre}(n)$, and $e \rightarrow f$ a production rule. The *safety condition* for n, x, and $e \to f$ is the disjunction of three parts (let g stand for a node and w for a guard):

1. $\neg e(x) \vee f(x) = x \vee n = f \vee$
2. $(n = g \wedge g(x) = x) \vee (n = w \wedge w(x)) \vee$
3. the stepwise execution of t_1 contains $exec(\mathsf{par}(\mathsf{ext}(n; \ldots), t, \ldots))$ where t contains f.

If this condition is satisfied for all n and x, the assignment f in t_2 cannot expose a difference between t_2 and t_1. (If firing a production rule exposes a difference, this is called a *misfiring*.)

Proof: Consider the usual stepwise evaluations of X_1 (with t_1) and X_2 (with t_2). Suppose in X_2 the next action is f in state x, and so far no differences have been exposed. Since t_1 is a non-terminating process, X_1 must be at a node or guard n of t_1, and $x \in pre(n)$. We consider each of the disjuncts of the above condition.

1a. If $\neg e(x)$, then, since we are at f, e must have changed from **true** to **false**. Because the production rules are stable, this can only have happened in a state x' where $f(x') = x'$. Even if the environment has changed x' to x, then still $f(x) = x$, because only a production rule, and not the environment, can change that (see the proof of Theorem 6.5). Hence, this case reduces to the next case.

1b. If $f(x) = x$, then performing the assignment does not expose a difference.

1c. If $n = f$, then execute f in both X_1 and X_2; no difference is exposed.

2a. If $n = g \wedge g(x) = x$, then execute g in X_1, reaching node or guard n'. Since execution of g does not change the state, $x \in pre(n')$, and the safety condition must hold for n' and x. Repeat this process until one of the other cases is reached.

2b. If $n = w \wedge w(x)$, then pass guard `[`w`]` in X_1, reaching node or guard n'. Since passing `[`w`]` does not cause a state change, $x \in pre(n')$. Repeat this process until one of the other cases is reached.

3. One of the possible interleavings is $ileave(t; \mathsf{ext}(n; \ldots), \ldots)$. Therefore, $x \in pre(t)$, and one of the other cases must apply. (It cannot be that case 3 applies to t, because we have excluded occurrences of $\mathsf{par}(s_1, s_2, \ldots)$ where s_1 and s_2 both contain assignment f.)

Since t_1 does not generate traces ending in ∞, there cannot be an infinite sequence of nodes and waits without state change; therefore case 2 always reduces to another case. If, in case 2 or 3, $e(x) \wedge f(x) \neq x$, then these cases always reduce to case 1c, where f is reached in X_1.

□

Suppose for some $x \in pre(n)$ the safety condition is not satisfied. Then, if n is a node g, it must be different from f and not vacuous. Since t_1 is a straight-line program, if g does not occur within a par, the only possible action in X_1 is g, so that firing f in X_2 exposes a difference. If g is part of a par, there may be several possible actions in X_1, but none equal to f (otherwise case 3 would apply), so that firing f exposes a difference. (It is not possible that $g \neq f \wedge g(x) = f(x)$, because g and f are constant assignments.) If n is a wait $[w]$, then apparently $\neg w(x)$, and $[w]$ cannot be passed. Since t_1 is a straight-line program, if $[w]$ does not occur within a par, no assignment from t_1 can be reached, so that firing f in X_2 exposes a difference. If $[w]$ occurs within a par, there may some assignments reachable, but none equal to f. We conclude that the safety condition is both sufficient and necessary to prevent misfirings by t_2. Note that this does not allow us to conclude $t_1 \leqslant_f t_2$, because t_2 may cause deadlock or have a fair trace ending in ∞.

We postpone that part of the proof, and first consider when t_1 can expose a difference. Let f be a node in t_1. We make the same assumptions about t_1 and t_2 as before. Suppose the following condition holds, which we refer to as the *progress condition*.

$$x \in pre(f) \Rightarrow f(x) = x \vee \langle \exists e \, : \, e \rightarrow f \text{ in } t_2 \, : \, e(x) \rangle$$

Then execution of f in t_1 cannot expose a difference between t_1 and t_2.

Proof: Consider X_1 and X_2, as before. Suppose f is reached in X_1 with state x, and no differences have been exposed. Assume in X_2 the execution is at a choice, but the choice has not yet been made (this assumption is valid because making the choice does not change the state). If $f(x) = x$, then executing f does not expose a difference. If $f(x) \neq x$, then let $e \rightarrow f$ be a production rule such that $e(x)$. Then clearly e can be passed in X_2 without state change, and f is reached in both X_1 and X_2, so that no difference is exposed.
□

Suppose that for some node f in t_1 the progress condition does not hold. Since $pre(f) \neq \{\}$, f can be reached in X_1, say with state x such that $f(x) \neq x$ and that there is no rule $e \rightarrow f$ with $e(x)$. Even if there is a rule of the form $e \rightarrow f$, the guard of this rule cannot be passed, so that f cannot be reached. (It is not possible that X_2 has already reached f, because, by the argument under 1a before, then $e(x) \vee f(x) = x$.) Hence, executing f in X_1 exposes a difference. Therefore, the progress condition is sufficient and necessary to prevent state changes in t_1 that expose a difference. Unlike the earlier case, this allows us to conclude that $t_1 \geqslant t_2$, because t_1 cannot cause deadlock or generate traces ending in ∞.

Consider the remaining cases for t_2. For t_2 to cause deadlock, all guards must be **false**. For t_2 to generate a fair trace ending with ∞, a situation must be reached where no state changes can occur, i.e., where all guards of effective (non-vacuous) production rules remain **false**. However, we know that t_1 cannot cause deadlock and will always eventually reach an effective assignment. Since $t_1 \geqslant t_2$, this assignment can also occur in t_2, implying that an effective production rule has a **true** guard. Hence, the progress condition prevents deadlock and fair traces without state changes. Therefore, the safety and progress conditions together are necessary and sufficient for $t_1 \equiv_f t_2$.

THEOREM 6.16 *(Production Rule Expansion)*

Let t_1 be a straight-line program such that there are no sync nodes, all guards are stable, and all assignments are constant assignments. Also assume that t_1

cannot cause deadlock or generate traces ending with ∞. Finally, assume that, if $\mathsf{par}(s_1, s_2, \ldots)$ occurs in t_1, then no assignment f occurs in both s_1 and s_2.
Let t_2 be a production rule set with stable production rules.

For node or guard n in t_1 and production rule $e \rightarrow f$ in t_2, define $\mathit{safety}(n, e \rightarrow f)$ as: (let g stand for an arbitrary node and w for an arbitrary guard)

$x \in \mathit{pre}(n) \Rightarrow$

1. $\neg e(x) \vee f(x) = x \vee n = f \vee$
2. $(n = g \wedge g(x) = x) \vee (n = w \wedge w(x)) \vee$
3. the stepwise execution of t_1 contains $\mathit{exec}(\mathsf{par}(\mathsf{ext}(n; s), t, \ldots))$ where t contains f.

For node f in t_1 define $\mathit{progress}(f)$ as:

$$x \in \mathit{pre}(f) \Rightarrow f(x) = x \vee \langle \exists e \,:\, e \rightarrow f \text{ in } t_2 \,:\, e(x) \rangle$$

Then,

$$\begin{aligned} &\langle \forall n, e, f \,:\, n \text{ a node or guard in } t_1 \wedge e \rightarrow f \text{ in } t_2 \,:\, \mathit{safety}(n, e \rightarrow f) \rangle \\ \wedge\, &\langle \forall f \,:\, f \text{ a node in } t_1 \,:\, \mathit{progress}(f) \rangle \end{aligned}$$

is necessary and sufficient for

$$t_1 \equiv_f t_2$$

□

Even if t_1 has the proper form (straight-line etc.), it may not always be possible to find a production rule set that satisfies the *safety* and *progress* conditions.

EXAMPLE 6.17

Let f and g be two different assignments in t_1 (as in Theorem 6.16) that do not occur within a par. Suppose there is a state x such that $x \in \mathit{pre}(f) \wedge x \in \mathit{pre}(g) \wedge f(x) \neq x \wedge g(x) \neq x$. Then $\mathit{progress}(f)$ requires a production rule $e \rightarrow f$ with $e(x)$, whereas $\mathit{safety}(g, e \rightarrow f)$ requires that $\neg e(x)$. Hence, there is no production rule set that satisfies both conditions. Since these conditions are necessary, the conclusion is that there is no production rule set t_2 (with stable guards) that is equivalent to t_1.

□

To implement a situation as in this example, extra state variables are inserted in t_1. For instance, if the sequence $f; \ldots g$ occurs, changing this to $u\uparrow; f; u\downarrow; \ldots g$ means that now $x \wedge u \in \mathit{pre}(f)$ whereas $x \wedge \neg u \in \mathit{pre}(g)$, so that the above problem does not occur. With sufficiently many state variables all such problems can be removed: Certainly, using this technique, all problems with the original assignments can be removed, by essentially implementing a program counter with state variables. The only interesting question is whether the state variables themselves may suffer from the same problem. Consider insertion of $u\uparrow$ and $u\downarrow$ such that $x \in \mathit{pre}(u\uparrow) \wedge x \in \mathit{pre}(u\downarrow)$. Then replace $u\uparrow; \ldots u\downarrow$ by

$$u\uparrow; v\uparrow; \ldots u\downarrow; v\downarrow$$

As a result, $x \wedge \neg v \in \mathit{pre}(u\uparrow)$, $x \wedge u \in \mathit{pre}(v\uparrow)$, $x \wedge v \in \mathit{pre}(u\downarrow)$, and $x \wedge \neg u \in \mathit{pre}(v\downarrow)$, so that all four states are different.

The following analysis shows that inserting state variables to distinguish between preconditions is sufficient to guarantee the existence of a production rule set. Define t_1 and t_2 as before. From the *progress* condition it follows that if $u\uparrow$ occurs in t_1, then there must be a production rule $e_u \rightarrow u\uparrow$ in t_2 (unless $u\uparrow$ is always vacuous in t_1). Assume that this is the only production rule for $u\uparrow$. *progress*($u\uparrow$) says that $x \in \mathit{pre}(u\uparrow) \Rightarrow x(u) \vee e_u(x)$. Hence,

$$x \in \mathit{pre}(u\uparrow) \wedge \neg x(u) \Rightarrow e_u(x)$$

Define guard e as $e(x) \equiv x \in \mathit{pre}(u\uparrow) \wedge \neg x(u)$; then we must have

$$e(x) \Rightarrow e_u(x)$$

Next, consider *safety*($n, e_u \rightarrow u\uparrow$). Let n be a node or guard in t_1, and let x be a state, such that $x \in \mathit{pre}(n) \wedge n \neq u\uparrow \wedge \neg x(u) \wedge \neg 2 \wedge \neg 3$ (where $\neg 2$ means case 2 from the definition of *safety* does not apply). Then *safety*($n, e_u \rightarrow u\uparrow$) $\Rightarrow \neg e_u(x)$. Any e_u that satifies this, satisfies the *safety* condition. In particular, from the earlier implication we have $\neg e_u(x) \Rightarrow \neg e(x)$. Hence, e satisfies the *safety* condition. Since e also satisfies the *progress* condition, it follows that e is a valid guard for $u\uparrow$. Furthermore, it follows that if e does not satisfy the *safety* condition, then neither does e_u (assuming $e(x) \Rightarrow e_u(x)$). Hence, we have the following corollary to Theorem 6.16.

COROLLARY 6.18

Let t_1 be defined as in Theorem 6.16. Suppose $u\uparrow$ is a, potentially effective (i.e., non-vacuous), assignment in t_1. There exists a single production rule $e_u \rightarrow u\uparrow$ for $u\uparrow$ (with stable guard) if and only if the guard

$$x \in \mathit{pre}(u\uparrow) \wedge \neg x(u)$$

satisfies the *safety* condition.

□

(Stability follows from Lemma 6.19 below.) In the above analysis, say *safety*($n, e \rightarrow u\uparrow$) does not hold for some node or guard n and state x. Then it must be that $x \in \mathit{pre}(u\uparrow) \wedge x \in \mathit{pre}(n) \wedge n \neq u\uparrow \wedge \neg x(u) \wedge \neg 2 \wedge \neg 3$. We have already shown how state variables can be inserted to make sure that $\mathit{pre}(u\uparrow) \cap \mathit{pre}(n) = \{\}$, which prevents this situation. Hence, insertion of state variables is sufficient to guarantee the existence of a production rule implementation.

Generally, we are only interested in production rules with stable guards, because stability is required by most VLSI implementations. The following lemma shows that it is straightforward to satisfy this requirement.

LEMMA 6.19

Let t_1 have the form defined in Theorem 6.16. Let t_2 be a production rule set where each gate has the form

$*[[\ e_u \longrightarrow\ u\uparrow\ ▯\ e_d \longrightarrow\ u\downarrow\ ]]$

If the *safety* and *progress* conditions described in Theorem 6.16 are satisfied, then all production rules in t_2 are stable.

Proof: Consider when e_u can be falsified; the case for e_d is identical. Let $e_u(x)$ for $x \in pre(n)$ for some node or guard n in t_1.

We know that $safety(n, e_u \rightarrow u\uparrow)$ holds. From the earlier proof we know that, after a number of steps, case 2 always reduces to another case without changing the state. Hence, if e_u can be falsified before these steps, it can also be falsified after these steps. Form the same proof, we also know that case 3 always reduces to another case (without any steps). Hence, we only need to consider case 1. Since we assume $e_u(x)$, we must have $u\uparrow(x) = x \vee n = u\uparrow$, i.e., $x(u) \vee n = u\uparrow$. If $x(u)$, e_u may be falsified without violating the stability requirement (Definition 6.4).

Assume $\neg x(u) \wedge n = u\uparrow$ (note that $\neg u$ can only be changed by firing $e_u \rightarrow u\uparrow$). From $progress(u\uparrow)$, it then follows that

$$\langle \exists e\ :\ e \rightarrow u\uparrow \text{ in } t_2\ :\ e(x)\rangle$$

However, there is only one rule $e \rightarrow u\uparrow$ in t_2, namely $e_u \rightarrow u\uparrow$. Hence, as long as the execution is in a state $x \in pre(n)$, $e_u(x)$ must continue to hold. The only step that can change this is execution of n itself, that is, execution of $u\uparrow$. That execution itself may falsify e_u without violating the stability requirement, and following the assignment u holds, so that e_u may be falsified as well.

□

This is a useful lemma: Suppose an algorithm is designed to generate production rules satisfying the *safety* and *progress* conditions. According to the lemma, this algorithm is then automatically guaranteed to only generate stable production rules.

Theorem 6.16 does not exclude interfering and self-invalidating production rules. Consider a pair of interfering production rules $e_u \rightarrow u\uparrow$ and $e_d \rightarrow u\downarrow$. The interference can be removed by strengthening the guards. In particular, $e_u \wedge \neg u \rightarrow u\uparrow$ and $e_d \wedge u \rightarrow u\downarrow$ have no interference; instead, these production rules are self-invalidating. Although, in general, this may not be the best strengthening of the guards (since it removes all vacuous firings), it shows that the problem of interfering production rules can be reduced to that of self-invalidating production rules.

Normally, a self-invalidating production rule can be changed to one that is not self-invalidating without affecting the conditions of the theorem. Suppose a self-invalidating production rule of the form $e_u \wedge \neg u \rightarrow u\uparrow$ is really needed to satisfy the conditions. Then apparently the conjunct $\neg u$ has been added to distinguish $e_u \wedge \neg u$ from a state where $e_u \wedge u$, to prevent a firing in the latter state. But in that state the firing would be vacuous. The only reason to exclude a vacuous firing is when the vacuous firing would cause interference. Hence, the only reason to use this guard is if $e_d \rightarrow u\downarrow$ can fire in state $e_u \wedge u$. But $u\downarrow$ would change that state to $e_u \wedge \neg u$, so that $u\downarrow$ can be immediately followed by $u\uparrow$. Hence, we conclude that the only reason to choose a self-invalidating rule for $u\uparrow$ is the presence of $u\downarrow; u\uparrow$ in the straight-line program. This is essentially the same as Example 6.7. The solution is to insert state variable assignments between $u\downarrow$ and $u\uparrow$. Observe also that after $u\downarrow; u\uparrow$ the same state holds as before this sequence, so that $u\downarrow$ can fire again. Therefore, the straight-line program must in fact contain sequences $u\uparrow; u\downarrow; u\uparrow; u\downarrow; \ldots$ of arbitrary length (the sequence can be terminated because of assignments by the environment). Hence, if the straight-line program does not contain such a sequence, self-invalidating production rules (and, therefore, interfering production rules) can always be avoided.

Chapter 7
Conclusion

7.1 Summary

In this text we have defined a new form of formal operational semantics. The semantics formally defines familiar concepts such as programs, environments, program execution, and computation. It goes beyond the usual formal operational semantics by defining an implementation relation (also called refinement relation) based on the result of program execution. We have shown how stepwise evaluation of the execution function (based on the structure of its definition) can be used to prove an implementation relationship between two programs. The definition of implementation is based on observation by an arbitrary environment. In particular, there is no need for special testing environments with success and failure states. This way, the semantics naturally includes the means to prove transformations that are only allowed in restricted contexts.

Because the semantics is operational and based on definitions of familiar concepts, the proofs can often follow the informal lines of reasoning that are commonly used in practice, thus making the formal method easier to use. Furthermore, the semantics directly supports many features found in programming languages, including sequential and concurrent composition, non-determinism, shared variables, infinite computations, and the simultaneous completion of synchronization actions. This makes it straightforward to express the semantics of languages that contain these features.

Although the semantics was developed to be used with a VLSI synthesis method, it is reasonably general and may be useful for the definition of general-purpose concurrent programming languages as well.

After defining the method of operational semantics, we have demonstrated its use as a semantic framework for Martin synthesis, a synthesis method for asynchronous VLSI circuits. We have defined the semantics of each of the languages used in the synthesis method, CSP, Handshaking Expansions (HSE), and Production Rule Sets (PRS). We have used the semantic framework to describe many of the method's transformations, and used the implementation relation to prove their correctness. This not only increases the confidence in the synthesis method, but may also give insight into the exact workings of transformations and help develop new transformations. For transformations that are not proven in this text, in particular, for the problem-specific transformations used in the early stages of a design, the semantic framework establishes the proof requirements.

Among the proven transformations are process decomposition, process factorization, handshake expansion, handshake reshuffling, choice decomposition, and production rule expansion. In particular the proof of handshake expansion is generic, and can be applied to many variations of handshake protocols. Using this generic proof, a new protocol (the one-phase handshake) was discovered.

Because of the generality of the semantic framework, which was designed with this application in mind, we could deal with the synthesis method in its full generality, without restrictions or assumptions which are not satisfied by actual designs. For instance, we allow the use of shared variables, local parallelism within a process, multiple handshake protocols, and arbitration.

7.2 Future Work

Although we introduced a fair implementation relation in Section 6.3, the fairness constraint it imposes is very weak. It would be useful if other fairness constraints could be included as well. Also, in Section 6.3 we add fairness by removing unfair traces from the trace set. Since fairness is a property of the interleaving, it would be better if the fairness constraint could be added to the interleaving function. That may not be straightforward, however, because the interleaving function performs a local transformation using only local information, whereas fairness is a global constraint.

The semantics includes data (variables and expressions), and most proven transformations do involve data aspects. Nevertheless, we have not really done data refinement, i.e., changing the representation of data, such as replacing an integer by its binary representation. Inclusion of such transformations is especially useful in the early stages of a design. Naturally, if the representation of variables in a trace is changed, the traces will be different, invalidating the implementation relation. One solution is to apply a mapping to the traces that replaces the new representation by its original form. A problem with that solution is that it often creates intermediate values that are not part of the original trace. For instance, an integer can be changed in a single assignment, but its binary representation must be changed one bit at a time, generating many intermediate values. Since an underlying assumption of the semantics is that the environment can observe any state, we cannot easily hide the intermediate values. A better solution is to only change the representation of local variables, and add assignments to the program that explicitly convert the representation before it is assigned to an environment variable. This method does not require any change of the semantics, but requires that data refinement is done in several steps.

We believe the semantics can be useful for general-purpose programming languages.

However, such applications may require some changes to the semantics. For instance, the current semantics does not easily support jumps such as **goto** or **return** statements. To add direct support for these statements, the tree ADT may need to be extended.

Finally, the proofs in this text form only the first application of the semantic framework. It would be useful to describe the complete compilation procedure using the notations developed in this text. Furthermore, it is worthwhile to attempt to prove, for specific designs, the problem-specific transformations that are part of the refinement stage.

Appendix A

Abstract Data Types

To describe our semantics, we need some 'new' data types. One way to define a new data type is to express it in terms of 'standard' data types, such as integers, sets, and functions. For instance, one can define a list of elements as a function from indices to elements. Theorems involving lists can then be proven by translating them to theorems about functions and proving those theorems using properties of functions. One might call a type defined in this way a concrete data type, because it is defined in terms of concrete mathematical objects. An alternative method of data type definition is to give a set of equations that specify the properties we want the new type to have. Theorems involving the data type can then be proven by using the properties expressed by the equations. A type defined with this method is called an abstract data type (ADT).

Concrete data types are suitable for simple types. Their main advantage is that many properties of the new type follow immediately from properties of the underlying data types. However, many new types are hard to map onto existing types. It then becomes hard to work with the new data type because of the complex translation to the underlying types. In those cases, abstract data types are preferred, because they do not involve such translations. In addition, abstract data types have the advantage that they don't rely on other types already being known. A well-known example of an abstract data type is the natural numbers as defined by the Peano axioms. In Chapter 2 we define traces and trees as abstract data types. In this appendix, we give a simple example of an ADT, and discuss some of the issues involved.

A.1 The List ADT

As an example, we define a type 'list of elements of E' as abstract data type. The defining equations are usually divided into 'constants and constructors,' which describe the form of elements of the type, and other axioms, which give additional properties of the type. In particular, axioms may be used to define when two elements are considered equal. Here is a simple set of equations (or axioms) defining the abstract data type **list of** (E).

Let *List* = **list of**(E). For any $e \in E$ and any $r, s, t \in List$, the following axioms hold.

- Constants and constructors

  L1. $\emptyset \in List$
  $e \in List$

  L2. $s +\!\!+\, t \in List$

- Equivalence axioms

  L3. $s +\!\!+\, \emptyset = s = \emptyset +\!\!+\, s$

  L4. $(s +\!\!+\, t) +\!\!+\, r = s +\!\!+\, (t +\!\!+\, r)$

(Parentheses are not considered part of the type definition; they have their usual meaning.)

Let us consider each axiom in turn. Axiom L1 says that there is a list '$\emptyset$'; we'll call this the empty list. The second part of L1 specifies that a single element is a list by itself. We could have chosen a different notation for single-element lists, such as '$[e]$.' But if there is usually no confusion between elements and single-element lists, or if the distinction is not important, we may as well keep the notation as simple as possible.

Axiom L2 says that, if s and t are lists, then $s+\!\!+t$ is a list as well. The '$+\!\!+$' constructor is called concatenation. In addition to the construction axioms, we make a hidden assumption that there are no other constants or constructors. From this assumption it follows that a list must have one of the three forms '$\emptyset$', 'e', or '$s +\!\!+\, t$.' This assumption allows us to use two important techniques, structural induction and structural definition. Structural induction is induction on the number of construction steps, i.e., the number of applications of L1 and L2 necessary to construct a list. Structural definition (or definition by structural induction) is a similar technique to define functions (or relations) on lists: the function value on a list is specified by specifying the function values on the component lists, which have fewer construction steps. An alternative to making this assumption is to add an axiom that explicitly specifies structural induction as a proof method. For convenience, we will always make the assumption, but not write it out explicitly.

Two lists are obviously equal if they are constructed in the same way. However, the equivalence axioms specify that there may be more than one way to construct the same list. In particular, L3 specifies that the empty list is the (left and right) neutral element of concatenation, and L4 says that concatenation is associative. What L3 and L4 really do is to define a relation '$=$,' which we intend to be an equivalence relation. Hence, we must assume that '$=$' is reflexive ($s = s$), transitive ($s = t \wedge t = r \Rightarrow s = r$), and symmetric ($s = t \Rightarrow t = s$), and that the constructors are monotonous over '$=$', i.e., $s = s' \wedge t = t' \Rightarrow s+\!\!+t = s'+\!\!+t'$. Again, we will not state these additional properties explicitly, but take it that the choice of the '$=$' symbol is sufficient indication of our intentions. Finally, we make the assumption that there are no other equivalence axioms, i.e., if $s = t$ then either

s and t are identically constructed, or they can be proven equal by the given axioms and assumptions. If we need to distinguish between the two types of equality, we will write s is t to mean that s and t are identically constructed.

A.2 Normal Forms

The equivalence axioms cause a problem when we want to define functions by structural definition. Say we define f as follows: $f(\emptyset) = \emptyset$, $f(e) = e$, $f(s \mathbin{+\!\!+} t) = t \mathbin{+\!\!+} s$. Unfortunately, with this definition, $s = t$ does not imply that $f(s) = f(t)$. For instance, $f((e_1 \mathbin{+\!\!+} e_2) \mathbin{+\!\!+} e_3) = e_3 \mathbin{+\!\!+} (e_1 \mathbin{+\!\!+} e_2)$, whereas $f(e_1 \mathbin{+\!\!+} (e_2 \mathbin{+\!\!+} e_3)) = (e_2 \mathbin{+\!\!+} e_3) \mathbin{+\!\!+} e_1$. Generally, we only want to define functions for which $s = t$ implies $f(s) = f(t)$. One solution is to prove for each function we define that this is the case (such proofs can be given because we know all rules that can prove two lists to be equal). Another method is to define a normal form. Choosing a normal form amounts to choosing a unique representative for each equivalence class of the '=' relation. If s and t are in normal form, i.e., they are such representatives, then $s = t$ implies s is t. If f is defined by structural definition, then certainly s is t implies $f(s) = f(t)$. Hence, if we specify that f can only be applied to lists in normal form, we automatically get the desired monotonicity:

$$s = t \Rightarrow s \text{ is } t \Rightarrow f(s) = f(t)$$

It is particularly convenient if we can specify representatives by specifying the form of their construction. In that case, a function f need only be defined by structural definition on the construction of lists in normal form.

The following is a possible normal form for lists.

Let $e \in E$ and $s \in$ **list of**(E). A list is in normal form (NF), if it has one of the following forms:

N1. $\emptyset$ or e

N2. $e \mathbin{+\!\!+} s$ where s is in NF and s is̸ $\emptyset$.

Hence, a list is in normal form if it is written in a right-associative way and without superfluous empty lists.

We must now prove that this is indeed a normal form, i.e., that each list can be written in normal form and that normal forms are unique. We restrict ourselves to finite lists.

- Lemma: If t and r are in NF, then $t \mathbin{+\!\!+} r$ can be written in NF.

  We prove this by structural induction on the form of t (which is described by rules N1 and N2). Let $s < t$ stand for 's has fewer construction steps than t.'

Basis: Assume t is described by N1. If t is $\emptyset$, then $t +\!\!+ r = r$, which is in NF. If t is e, then $t +\!\!+ r$ is in NF if r is $\not\equiv \emptyset$; otherwise, if r is $\emptyset$, then $t +\!\!+ r = t$, which is in NF.
Induction: The induction hypothesis (IH) is: for any $s < t$: if s is in NF then $s +\!\!+ r$ can be written in NF. We prove that $t +\!\!+ r$ can be written in NF.

Let t is $e +\!\!+ t'$, where t' is in NF. Then

$$t +\!\!+ r \text{ is } (e +\!\!+ t') +\!\!+ r = e +\!\!+ (t' +\!\!+ r)$$

Since $t' < t$, $t' +\!\!+ r$ can be written in NF, say as s (which is not $\emptyset$). Then $t +\!\!+ r = e +\!\!+ s$, which is in NF.

- Every finite s can be written in NF. Again, we use structural induction; this time, the form of s is described by L1 and L2.
  Basis: If s is constructed by L1, s is in NF.
  Induction: The IH is: every $s' < s$ can be written in NF.

  Say s is $t +\!\!+ r$. According to the IH, t and r can be written in NF, say as t' and r' respectively. Then $s = t' +\!\!+ r'$, which, according to the above lemma, can be written in NF.

We still need to prove that normal forms are unique. For this proof we need to use the implicit assumption that there are no other ways of proving $s = t$ than with the given axioms. For convenience, we give two consequences of this assumption, without proof (these rules could have been given as additional axioms):

For $e, f \in E$ and $s, t \in$ *List*:

- Uniqueness axioms

  L5. $e +\!\!+ s = f +\!\!+ t \Rightarrow e = f \wedge s = t$

  L6. $e +\!\!+ s \neq \emptyset$

- Proving uniqueness of the normal form corresponds to proving

$$\text{If } s \text{ and } t \text{ are in NF, then } s = t \Rightarrow s \text{ is } t.$$

Since we are only concerned with the normal form for lists, we assume that elements from E are unique (or that they are written in a suitable normal form). We use structural induction on the form of s. Assume $s = t$; in each case, we show that either this is a contradiction, or that s is t.
Basis: s is described by N1.

Let s is $\emptyset$.

(1) t is $\emptyset$: then s is t.

(2) t is e: $t = e = e +\!\!+ \emptyset \neq \{L6\}\ \emptyset = s$, so $s \neq t$.

(3) t is $e +\!\!+ t'$: again, by L6, $t \neq \emptyset$ and hence $s \neq t$.

Hence, $\emptyset$ is unique in NF.

Let s is e.

(1) t is $\emptyset$: as (2) above.

(2) t is f: elements from E are unique, so $e = f \Rightarrow e$ is f and hence s is t.

(3) t is $f +\!\!+ t'$, where t' is in NF and t' is̸ $\emptyset$: From $s = t$ it follows that $s = e = e +\!\!+ \emptyset = f +\!\!+ t'$, from which we get, by L5, that $t' = \emptyset$. But since $\emptyset$ is unique in NF, either t' is $\emptyset$ or t' is not in NF, both contradictions.

Induction: The IH is: if $p < s$ and if p and q are in NF, then $p = q \Rightarrow p$ is q.

Assume $s = e +\!\!+ s'$, with s' in NF. The cases t is $\emptyset$ and t is f lead to contradictions, as shown above. Hence, assume t is $f +\!\!+ t'$. From $s = t$, it then follows (by L5 and uniqueness of E) that e is f and $s' = t'$. But $s' < s$, so according to the IH we have s' is t', and hence s is t.

The proofs showing that N1 and N2 do indeed describe a normal form are not difficult, but rather elaborate. And since the proofs are based on structural induction, they increase in size as there are more constructors. Therefore, elsewhere in this thesis, we will present normal forms, but not the proofs that they are indeed that. Also, we will not explicitly state uniqueness axioms, but make the implicit assumption that elements are only equal if that can be proven with the given axioms.

The above proofs were restricted to finite lists because we cannot simply use structural induction to talk about infinite lists. In this thesis, we do allow infinite data structures, i.e., objects that are constructed from infinitely many constructors. A method to specify an infinite object is to define it as the solution of some recursive equation (often, as the least fixpoint of an equation). In that case, the axioms should be extended to postulate that such a solution indeed exists. However, in this thesis we are not particularly interested in the technicalities arising from the infinite nature of some objects; on the contrary, generally we are only interested in finite parts of objects. Since furthermore our infinite structures tend to be quite regular, we treat them somewhat informally, by saying that infinite objects exist and by writing them with a '...' notation. E.g., we could write $e +\!\!+ (e +\!\!+ (e \ldots) \ldots)$ to denote a particular infinite list and deal with it by looking at all its finite prefixes.

As a final word about normal forms we should remark that there are several alternative ways of defining ADTs that avoid them. First, we could consider only the 'is' relation as equality, and have L3 and L4 define a new relation '$\sim$' instead of '$=$'. Then we could for each function on the ADT consider separately whether it is monotonous over '$\sim$' or not. Second, we could attempt to define the constructors in such a way that the ADT consists of only elements in normal form. (In fact, that is how lists are usually defined in programming languages.) General concatenation could then be defined as a function on lists. Each of the methods has its pros and cons, and the choice is to a certain extent arbitrary. Incidentally, the method of choosing a normal form should be clear: mainly, it consist of applying each axiom in one direction, until it can no longer be applied.

A.3 Models

The specification of an abstract data type is just a set of equations. Hence, it could be inconsistent, meaning that it does not have a solution. The easiest way of showing consistency is by giving a solution, called a *model*. A model should be described in terms of an 'accepted' theory, such as set theory. (When the accepted theory is a programming language, a model is usually called an *implementation* of the ADT.) Hence, a model of an abstract data type is a concrete data type that satisfies the given axioms. The existence of a model for an ADT shows that the ADT is at least as consistent as the theory in which the model is described.

As an example, we give a concrete data type that models the lists defined above. We model a list by a pair $\langle n, f\rangle$ where n is a natural number and f is a function mapping from the interval $[0\ldots n)$ to the element type E. ($[0\ldots n)$ is the subset of $\mathbb{N}$ starting with 0 and ending with $n-1$.)

$$\langle n \in \mathbb{N}, f : [0\ldots n) \rightarrow E\rangle$$

For L1 we have to specify how the lists $\emptyset$ and e are modeled. We assume functions have the proper domain; function '$\emptyset$' has empty domain.

$$\begin{aligned} \emptyset &= \langle 0, \emptyset\rangle \\ e &= \langle 1, e\rangle \end{aligned}$$

To model L2, we must express the concatenation constructor as an operation on pairs $\langle n, f\rangle$:

$$\langle s, f\rangle +\!\!+ \langle t, g\rangle = \left\langle s+t, \begin{cases} f(x) & \text{if } x \in [0\ldots s) \\ g(x-s) & \text{if } x \in [s\ldots s+t) \end{cases}\right\rangle$$

As can be seen, both f and g are only applied to elements from their respective domains. Next we must prove that axioms L3 and L4 are indeed satisfied. Each of these proofs is straightforward; as an example, we give the first part of L3:

$$\langle s, f\rangle +\!\!+ \langle 0, \emptyset\rangle = \left\langle s+0, \begin{cases} f(x) & \text{if } x \in [0\ldots s) \\ \emptyset(x-s) & \text{if } x \in [s\ldots s+0) \end{cases}\right\rangle =$$

$$\left\langle s, \begin{cases} f(x) & \text{if } x \in [0\ldots s) \\ \emptyset(x-s) & \text{if } x \in \{\} \end{cases}\right\rangle = \langle s, f\rangle$$

Likewise, the uniqueness axioms can be proven, for instance L6:

$$\langle 1, e\rangle +\!\!+ \langle s, f\rangle = \left\langle s+1, \begin{cases} e & \text{if } x = 0 \\ f(x-1) & \text{if } x \in [1\ldots s+1) \end{cases}\right\rangle \neq \langle 0, \emptyset\rangle$$

Although models have their uses (e.g., they can help 'visualize' properties of the ADT), we generally do not use them in our calculations. Therefore, in chapter Chapter 2 we only give an informal sketch of the models, omitting their precise specifications and proofs.

A.4 Notes

In practice, often a model is known, and an ADT is designed to formally express the properties of that model. Such axiomatic definitions are common in mathematics, dating all the way back to Euclid (300 BC). A well-known example of an ADT is the natural numbers as defined by the Peano axioms (in 1889). That axiomatization does not use our implicit assumptions, but explicitly states the principle of induction as well as uniqueness axioms. Also, it does not contain equivalence axioms, meaning that all elements of the ADT can be considered to be in normal form (as described in the second alternative at the end of section Section A.2). A common model for the Peano axioms is provided by (abstract) set theory, by equating number 0 with the empty set $\{\}$, and the successor of n with $n \cup \{n\}$. Our treatment of ADTs is not as thorough and precise as the usual axiomatizations in mathematics. We feel this is justified because we do not intend to perform an in-depth analysis of the types we introduce; rather, we give ADTs to justify our use of the new types in the definition of our semantics.

The term 'abstract data type' is commonly used in computer science. However, there it is sometimes confused with concrete data types that are expressed in standard mathematics, as contrasted with implementations that are expressed in a programming language; as mentioned above, an implementation is just a model of an ADT. Programming language support for the use of ADTs first appeared in Simula 67, and forms the core of what is now called object-oriented programming.

Bibliography

[1] R.J.R. Back, K. Sere
Stepwise refinement of parallel algorithms
Science of Computer Programming 13, 133–180, 1990

[2] J.A. Bergstra, J.W. Klop
Process algebra for synchronous communication
Information and Control, 60, 109–137, 1984

[3] S.M. Burns
Performance Analysis and Optimization of Asynchronous Circuits
Ph.D. thesis, CS-TR-91-01, California Institute of Technology, 1991

[4] K.M. Chandy, J. Misra
Parallel Program Design: A Foundation
Addison-Wesley, Reading, MA, 1988

[5] E.W. Dijkstra, C.S. Scholten
Predicate Calculus and Program Semantics
Springer-Verlag, New York, NY, 1990

[6] J.C. Ebergen
A formal approach to designing delay-insensitive circuits
Distributed Computing, 5, 107–119, 1991

[7] N. Francez
Fairness
Springer-Verlag, New York, 1986

[8] C.A.R. Hoare
Communicating Sequential Processes
Communications of the ACM, 21:8, 666–677, 1978

[9] C.A.R. Hoare
Communicating Sequential Processes
Prentice-Hall International, Englewood Cliffs, N.J., 1985

[10] M.B. Josephs, J.T. Udding
An algebra for delay-insensitive circuits
pp. 147–175 in: E.M. Clarke, R.P. Kurshan, editors
Computer-Aided Verification '90
International Workshop on Computer Aided Verification
DIMACS series, 3
American Mathematical Society, Providence, RI, 1991

[11] T.K. Lee
A General Approach to Performance Analysis and Optimization of Asynchronous Circuits
Ph.D. thesis, California Institute of Technology, 1995

[12] A.J. Martin
The probe: An addition to communication primitives.
Information Processing Letters, 20:3, 125–130, 1985

[13] A.J. Martin
The design of a self-timed circuit for distributed mutual exclusion
pp. 247–260 in: H. Fuchs, editor
1985 Chapel Hill Conference on VLSI
Computer Science Press, Rockville, MD, 1985

[14] A.J. Martin
Programming in VLSI: From communicating processes to delay-insensitive circuits.
Chapter 1 in: C.A.R. Hoare, editor
Developments in Concurrency and Communication
UT Year of Programming Series
Addison-Wesley, Reading, MA, 1990

[15] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus
The design of an asynchronous microprocessor
pp. 351–373 in: C.L. Seitz, editor
Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI
MIT Press, Cambridge, MA 1989

[16] R. Milner
Communication and Concurrency
Prentice-Hall International, Englewood Cliffs, N.J., 1989

[17] P.D. Mosses
Action Semantics
Cambridge University Press, Cambridge, 1992

[18] C.D. Nielsen, A.J. Martin
Design of a delay-insensitive multiply-accumulate unit
Integration, the VLSI Journal 15:3, pp. 291-311, 1993

[19] G.D. Plotkin
A Structural Approach to Operational Semantics
Report no. DAIMI FN-19, Aarhus University, C.S. Department, 1981

[20] G.D. Plotkin
An operational semantics for CSP
pp. 250–252 in: A. Salwicki
Logics of Programs and Their Applications
Lecture Notes in Computer Science 148
Springer-Verlag, Berlin, 1983

[21] C.L. Seitz
System Timing
Chapter 7 in: C.A. Mead, L.A. Conway
Introduction to VLSI Systems
Addison-Wesley, Reading, MA, 1980

[22] S.F. Smith, A.E. Zwarico
Provably correct synthesis of asynchronous circuits
pp. 237–260 in: J. Staunstrup, R. Sharp, editors
Designing Correct Circuits
Formal Methods in System Design, 3:3 and 4:1

[23] J.L.A. van de Snepscheut
Trace Theory and VLSI Design
Lecture Notes in Computer Science, 200
Springer-Verlag, Berlin, 1985

[24] J.L.A. van de Snepscheut, J.T. Udding
An alternative implementation of communication primitives
Information Processing Letters, 23, 231–238, 1986

[25] J.A. Tierno, A.J. Martin, D. Borkovic, T.K. Lee
A 100-MIPS GaAs asynchronous microprocessor
IEEE Design & Test of Computers, 11:2, 43–49, 1994

[26] J.A. Tierno
An energy complexity model for VLSI Computations
Ph.D. thesis, CS-TR-95-02, California Institute of Technology, 1995