Bachelor's Thesis Computing Science

Semantics of Languages with Goto

Rutger Dinnissen
s4548337

October 24, 2023

First supervisor/assessor:
prof. dr. Herman Geuvers

Second assessor:
dr. Freek Wiedijk

Radboud University

Abstract

The semantics of a programming language can be used to reason about the behavior of a program. This makes it possible to prove that some program has the desired outcome, or even that compilations between programming languages are correct. While by far most programs are written in higher-level languages, those programs must be compiled to machine code, so that the program can be executed. Machine code, like other low-level languages, has goto (or jump) statements, which are harder to reason about. Even so, semantics of goto languages are needed, if compilers are to be proven correct by preserving semantics. This paper explains how to reason about the semantics of goto languages, and why certain constructs in the semantics are necessary.

Contents

Chapter 1

Introduction

The world of programming languages can be split into two categories: languages with goto (or jump) statements, and languages without those. While the most common high-level programming languages do not contain these kinds of control flow features, machine code does. Thus every program of any language will inevitably be compiled to a language with goto statements. Thus, reasoning about the semantics of goto statements is a necessity to prove that compilations preserve semantics. That being said, one has argued that goto statements make a programming language inherently non-modular, and thus hard to reason about. Dijkstra even said that he is "convinced that the go to statement should be abolished from all 'higher level' programming languages" [4].

However, one has argued that the difficulty of understanding programs with goto is not inherent to the goto statement itself. Rubin argues that programs can be shorter and less complex when using goto statements: "I introduce GOTOs to untangle each deeply nested mess of code, I have found that the number of lines of code drops by 20-25 percent" [8]. Furthermore, goto statements are an essential part of machine code, so at least when constructing a compiler, one should understand how the goto statement works.

Around the year 1970, the concepts for axiomatic and operational semantics emerged, but semantics for a language with goto statements were only defined a decade later, notably in 1981 by de Bruin in [3]. Natural semantics were first described in 1987 by Kahn in [5], but (compositional) natural semantics for a language with goto statements was only defined in 2005 by Saabas and Uustalu in [9].

One reason for it being difficult to define semantics for languages with goto statements is the fact that pieces of code (may) have multiple entry and exit points. The exit points are the goto statements, and the entry points are the labels.

In this thesis, we introduce the model language Goto, which is based on the

language While from one of the standard books on semantics by Nielson and Nielson [6]. Goto is essentially the same language used by de Bruin in [3]. We define structural operational semantics, natural semantics, and axiomatic semantics for Goto. For each of these approaches to formal semantics, we have a simple example to show that the interpretation of the goto statement is what one would expect, one example of an infinitely looping program, as well as one example for a program that computes the factorial of a number.

The chapter about structural operational semantics (SOS) is based on the work of de Bruin from [3], but changed to fit Goto and the notation used by Nielson and Nielson from [6]. We explain why certain choices have been made by de Bruin to define SOS rules for Goto. These choices are specific to the fact that we are working with a language with goto statements. As an example, the SOS rule for compositional statements in While is this:

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}\ [\text{comp}]^1_{\text{sos}}$$

Which basically means that for every composition $S_1; S_2$, do one step in S_1 and compose the resulting statement S_1' with S_2 again. However, this does generally not work, if S_1 is a goto statement, since then the program might jump over S_2 entirely, as shown in Section 4.1.1. SOS is also proven to be deterministic.

The ideas for the natural semantics (NS) we describe for Goto are based on ideas by Saabas and Uustalu from [9], as well as Nielson and Nielson from [6]. However, the language used by Saabas and Uustalu is different from our Goto in multiple subtle ways, which makes their ideas less straightforward to apply to Goto. One of the two most important factors is that Saabas and Uustalu do not make use of an if-then-else statement, but a conditional jump statement (`ifnot` b `then goto` l). This results in the conditional statement being primitive, and not compositional. Saabas and Uustalu make use of this by giving each primitive statement in a program its own label. They make use of this fact when defining their NS, but we show that their technique can also be applied to our Goto language. One of the NS rules we made for Goto is this:

$$\frac{\langle S_{pc},\ s\rangle \to \langle pc',\ \bullet,\ s''\rangle \qquad \langle P,\ pc',\ s''\rangle \to s'}{\langle P,\ pc,\ s\rangle \to s'}\ [\text{find}^{\bullet}_{\text{ns}}]$$

Here, P is the entire program, and pc is the program counter, which points to the next label, and S_{pc} is the code at label pc. $\langle S_{pc}, s\rangle \to \langle pc', \bullet, s''\rangle$ means that when executing statement S_{pc} in state s, we have encountered a goto statement, which jumps the program to label pc'.

Like for SOS, we explain in Chapter 5 how we arrived and why we defined NS for Goto the way we did. The same example programs we used

in the SOS chapter also have an NS interpretation. We prove that NS is deterministic, and claim that it is equivalent to the SOS we defined.

As for axiomatic semantics (AS), we use the same ideas by de Bruin from [3] to construct our AS rules. Usually, AS makes use of Hoare triples, which have the following form:

$$\{ P \}\ S\ \{ Q \}$$

where P and Q are both predicates and represent the pre- and postconditions of the Hoare triple and S is a statement of the program. De Bruin has added a dictionary to these Hoare triples like this:

$$\langle D \mid \{ P \}\ S\ \{ Q \} \rangle$$

The dictionary D is a list of pairs of labels and predicates (e.g. $D = n_1 : P_1, n_2 : P_2$). It is used to keep track of the preconditions for every statement corresponding to a label in the program. With this, it is possible to construct the rule

$$\langle D \mid \{ P \}\ \texttt{goto}\ n\ \{ \text{ false } \} \rangle$$

which has the condition of the precondition P being equal to the precondition corresponding to label n (i.e. the pair $n : P$ is included in the dictionary D). The postcondition is false, so that the precondition of the following statements can be met, as false implies every possible predicate. This is explained in more detail in Section 6.1. Like for SOS and NS, we show how the AS is applied with some examples. One of these examples is given on a step-by-step basis, to further show how to apply AS rules. We claim that AS is sound and complete with respect to the SOS described in this thesis.

The structure of this thesis is as follows:

First, the preliminaries in Chapter 2 give an overview of the work done by Nielson and Nielson about the language While and how structural operational semantics, natural semantics, and axiomatic semantics are applied to While in their work [6].

In Chapter 3 we describe the model language Goto, which we use for the rest of this thesis. It is based on While and in essence the same language as described by de Bruin in [3].

Then, Chapter 4 describes structural operational semantics for Goto, which is based on the work by de Bruin from [3].

This is followed up by defining natural semantics for Goto (based on ideas by Saabas and Uustalu from [9]) in Chapter 5.

Afterwards, axiomatic semantics for Goto are defined in Chapter 6. These semantics are the same as described by de Bruin from [3].

We conclude by mentioning some related work, before briefly summarizing this thesis.

Chapter 2

Preliminaries

2.1 While

While is the model language used by Nielson and Nielson in [6] to show different semantic approaches. This chapter summarizes their work but does not add anything new to it.

Most notably, a clear distinction between the syntax of a programming language and its semantics (i.e. meaning) is made. One example of this is the difference between syntactic numerals (like `1101`) and integers (like **13**).

Another example is the Boolean expressions `true` and `false`, and the actual Boolean values **tt** and **ff**. So **tt** means "true", and **ff** means "false". The words `true` and `false` are just considered syntax and have no inherent meaning.

The semantic functions define the relation between syntax and semantics as a function from the former to the latter.

2.1.1 Syntax of While

The syntax includes numeric, Boolean, and arithmetic expressions (**Num**, **Bexp**, and **Aexp**), as well as statements ($\mathbf{Stm}_{\mathrm{W}}$). The meta-variables n, b, a, and S range over these respective syntactic categories. Furthermore, the meta-variable x ranges over syntactic variable names **Var**. The most common variable names are `x`, `y`, and `z`, but in general, the set of variable names includes any string of alphabetical characters.

Note. The category $\mathbf{Stm}_{\mathrm{W}}$ is the only category with a subscript "W" to denote statements for the language While. The reason is that this paper introduces the language Goto, which uses the same syntax for numeric, Boolean, and arithmetic expressions, as well as the same variables. But since it uses a different syntax beyond that, we use a subscript letter to indicate what category of statements is meant.

Num	$n ::= \texttt{0} \mid \texttt{1} \mid n\,\texttt{0} \mid n\,\texttt{1}$
Bexp	$b ::= \texttt{true} \mid \texttt{false} \mid b_1 = b_2 \mid b_1 \leq b_2 \mid \neg b \mid b_1 \wedge b_2$
Aexp	$a ::= x \mid n \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$
$\mathbf{Stm}_{\mathrm{W}}$	$S ::= x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$
	$\mid \texttt{while}\ b\ \texttt{do}\ S$

Table 2.1: Syntax of the language While [6]

Definition 2.1.1 (Syntax of While)**.** The definition of the syntax of While can be found in Table 2.1.

2.1.2 States

The values of variables are stored in a state s. A state $s \in \mathbf{State}$ is a total function from variable names (**Var**) to integers ($\mathbb{Z}$).

As an example, if we want to define a state where the value of the variable x is 5, then we can say "Let s be a state with $s\ x = 5$". In this case, the values of other variables are unknown. For these cases, we write $s\ y = \perp$. Note that this does not mean that the variable y is undefined for this state.

Substitutions, which represent changes in a state, are written like this: $s[x \mapsto 5]$. In this case, s can be any state, thus in particular, the value of x is unknown. However, the substitution $s[x \mapsto 5]$ results in a state, where the value of x is 5. Thus $s[x \mapsto 5]\ x = 5$.

To make states more readable, we use the following notation to abbreviate substitutions:

$$\begin{aligned} s_a &= s[x \mapsto a] \\ s_{a,b} &= s[x \mapsto a][y \mapsto b] \\ s_{a,b,c} &= s[x \mapsto a][y \mapsto b][z \mapsto c] \end{aligned}$$

Where a, b, and c are integers. In case some variable is unknown, we use the $\perp$ symbol. As an example, all states s where $s\ y = 0$ can be written as $s[y \mapsto 0]$, or abbreviated with $s_{\perp,0}$.

2.1.3 Semantic functions

The semantics of the numerical, Boolean, and arithmetic expressions are defined with the semantic functions $\mathcal{N}$, $\mathcal{B}$, and $\mathcal{A}$, respectively.

Definition 2.1.2. The semantic function $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ for the interpretation of numerals can be found in Table 2.2.

$$
\begin{aligned}
\mathcal{N}[\![\texttt{0}]\!] &= \mathbf{0} \\
\mathcal{N}[\![\texttt{1}]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ \texttt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ \texttt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}
$$

Table 2.2: Semantic functions $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ for numerical expressions [6]. Integers are written in bold to make the difference between them and syntax more clear

$$
\begin{aligned}
\mathcal{A}[\![x]\!]s &= s\ x \\
\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![a_1\ \texttt{+}\ a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1\ \texttt{*}\ a_2]\!]s &= \mathcal{A}[\![a_1]\!]s\ \cdot\ \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1\ \texttt{-}\ a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{aligned}
$$

Table 2.3: Semantic function $\mathcal{A} : \mathbf{Aexp} \to \mathbf{State} \to \mathbb{Z}$ for arithmetic expressions [6]

Note. This definition of numerals only allows binary number representations. This is done to show the interpretation of numerals more clearly. However, for the sake of readability, we will make use of decimal numbers in the examples in this paper. Their interpretation is what one would expect, i.e. $\mathcal{N}[\![\texttt{37}]\!] = \mathbf{37}$.

Definition 2.1.3. The semantic function $\mathcal{A} : \mathbf{Aexp} \to \mathbf{State} \to \mathbb{Z}$ for the interpretation of arithmetic expressions can be found in Table 2.3.

Definition 2.1.4. The semantic function $\mathcal{B} : \mathbf{Bexp} \to \mathbf{State} \to \{\mathbf{tt}, \mathbf{ff}\}$ for the interpretation of Boolean expressions can be found in Table 2.4.

Note. To make a more clear distinction between the syntactical and semantic domains, syntax arguments are always passed in between the $[\![$ and $]\!]$ brackets. Consider the definition of interpreting the arithmetic expression of addition:

$$\mathcal{A}[\![a_1\ \texttt{+}\ a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$$

Here, the `+` on the left side is syntax and thus does not have an inherent meaning. However, the $+$ on the right side is outside the $[\![\ ]\!]$ brackets, thus this is the actual addition operator in the domain of semantics.

$$\mathcal{B}[\![\texttt{true}]\!]s = \mathbf{tt}$$
$$\mathcal{B}[\![\texttt{false}]\!]s = \mathbf{ff}$$
$$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases}$$
$$\mathcal{B}[\![a_1 \leq a_2]\!]s = \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases}$$
$$\mathcal{B}[\![\neg b]\!]s = \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases}$$
$$\mathcal{B}[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}$$

Table 2.4: Semantic function $\mathcal{B} : \mathbf{Bexp} \to \mathbf{State} \to \{\mathbf{tt}, \mathbf{ff}\}$ for Boolean expressions [6]

2.1.4 Semantic rules

The semantics of statements work a bit differently from the semantics functions described so far. There are multiple approaches for interpreting some statement S. In this paper, we focus on the approaches of operational semantics (in which we look at structural operational semantics and natural semantics) and axiomatic semantics.

Each form of semantics has its own set of rules for statements. Every rule has one conclusion.

Some rules have conditional statements, which need to hold to be able to apply that rule.

Some rules have premises. These are written above the conclusion, with a horizontal line separating the conclusion and the premises. Multiple premises of the same rule are separated by white space.

Rules that do not have premises do not have a horizontal line and are called *axiom schemes*.

Structural operational semantics

Structural operational semantics (SOS) is a step-by-step approach for interpreting the meaning of a statement.

For example, consider the statement `if x=0 then skip else x:=0`. Depending on the current value of `x`, the next step is either `skip` or `x:=0`. So

$$[\text{ass}_{\text{sos}}] \qquad \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$[\text{skip}_{\text{sos}}] \qquad \langle \texttt{skip}, s\rangle \Rightarrow s$$

$$[\text{comp}^1_{\text{sos}}] \qquad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$[\text{comp}^2_{\text{sos}}] \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[\text{if}^{\text{tt}}_{\text{sos}}] \qquad \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{sos}}] \qquad \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\text{while}_{\text{sos}}] \qquad \langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow \langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle$$

Table 2.5: Structural operational semantics for While [6]

in general, to be able to interpret a statement, we need to know what the current state of the program is.

The interpretation of any statement S with some state s is a *derivation sequence*, where each step is the application of one rule. Transitions are of the form $\langle S, s\rangle \Rightarrow \langle S', s'\rangle$ or $\langle S, s\rangle \Rightarrow s'$, depending on the rule.

$\langle S, s\rangle$ and s are both called *configurations*, while s in particular is also a *final* or *terminal configuration*. If no rule can be applied to some configuration $\langle S, s\rangle$, then it is called *stuck*.

Each transition is the conclusion of a derivation tree. However, in SOS, we generally do not keep track of any derivation trees. The existence of a derivation sequence $\langle S, s\rangle \Rightarrow \langle S', s'\rangle \Rightarrow \langle S'', s''\rangle \Rightarrow \ldots$ implies that there is a derivation tree with $\langle S, s\rangle \Rightarrow \langle S', s'\rangle$ as its conclusion.

We can write $\gamma \Rightarrow^p \gamma'$, to denote the existence of a derivation sequence from γ to γ', which is p steps long.

We can write $\gamma \Rightarrow^+ \gamma'$, to denote the existence of a derivation sequence from γ to γ', which is at least one step long.

We can write $\gamma \Rightarrow^* \gamma'$, to denote the existence of a derivation sequence from γ to γ', which is at least zero steps long. This means that it is possible that $\gamma = \gamma'$.

Definition 2.1.5 (SOS for While)**.** The structural operational semantics for While are defined in Table 2.5.

Natural semantics

Natural semantics (NS) is also based on transitions from a configuration with a statement and a state. The difference is that the transition directly results

$$[\text{ass}_{\text{ns}}] \qquad \langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$[\text{skip}_{\text{ns}}] \qquad \langle \texttt{skip}, s\rangle \to s$$

$$[\text{comp}_{\text{ns}}] \qquad \frac{\langle S_1, s\rangle \to s'' \quad \langle S_2, s''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'}$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \qquad \frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \qquad \frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\text{while}^{\text{tt}}_{\text{ns}}] \qquad \frac{\langle S, s\rangle \to s'' \quad \langle \texttt{while } b \texttt{ do } S, s''\rangle \to s'}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{while}^{\text{ff}}_{\text{ns}}] \qquad \langle \texttt{while } b \texttt{ do } S, s\rangle \to s \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Table 2.6: Natural semantics for While [6]

in some final state. Thus all transitions in NS are of the form $\langle S, s\rangle \to s'$. Applying NS rules results in a *derivation tree*.

Infinite derivation trees do not exist. If the interpretation of $\langle S, s\rangle$ requires an infinite tree, then there exists no state s' such that $\langle S, s\rangle \to s'$. This means that if S is executed in state s, then the program will not terminate.

Definition 2.1.6 (NS for While)**.** The natural semantics for While are defined in Table 2.6.

Axiomatic semantics

Axiomatic semantics (AS) is about proving partial correctness of a statement with respect to two predicates: the precondition and postcondition. If one were to execute a program S in a state that makes the precondition true, and the program terminates, then the final state makes the postcondition true. Such assertions are written as a so-called Hoare triple: $\{\ P\ \}\ S\ \{\ Q\ \}$.

Here, P and Q are predicates, which are functions from **State** to $\{\mathbf{tt}, \mathbf{ff}\}$. As an example, let $P := (x{=}5)$. Then $P\ s_5 = (5{=}5) = \mathbf{tt}$, and $P\ s_0 = (0{=}5) = \mathbf{ff}$.

Applying axiomatic semantic rules results in an *inference tree*. Such a tree only proves that if P holds in the initial state, then Q holds in the final state if the program terminates. The inference tree does not prove termination, hence only partial correctness can be proven.

Definition 2.1.7 (AS for While)**.** The axiomatic semantics for While are defined in Table 2.7. Rule names have the letter "p" subscripted to denote partial correctness.

$$[\text{ass}_\text{p}] \quad \{\ P[x \mapsto \mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ P\ \}$$

$$[\text{skip}_\text{p}] \quad \{\ P\ \}\ \texttt{skip}\ \{\ P\ \}$$

$$[\text{comp}_\text{p}] \quad \frac{\{\ P\ \}\ S_1\ \{\ Q\ \} \qquad \{\ R\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ S_1; S_2\ \{\ Q\ \}}$$

$$[\text{if}_\text{p}] \quad \frac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ Q\ \} \qquad \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}}$$

$$[\text{while}_\text{p}] \quad \frac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}}$$

$$[\text{cons}_\text{p}] \quad \frac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}$$
$$\text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q$$

Table 2.7: Axiomatic semantics for While [6]

Note. Rules like $[\text{ass}_\text{p}]$ are defined with $\mathcal{A}[\![a]\!]$, instead of $\mathcal{A}[\![a]\!]s$. This is done, because predicates are themselves evaluated in a state. As a result, there is no need to carry over and change states in the rules for AS.

Chapter 3

Goto

The most important part of a language with goto is the inclusion of labels L, as well as some form of a `goto` L statement.

Including some conditional statement like `if` b `then` S_1 `else` S_2 or even just a conditional jump statement like `if` b `then goto` L will make the language Turing complete.

3.1 Syntax of Goto

Goto is designed to be as similar to While as possible. For this reason, the syntax for numerals, as well as Boolean and arithmetic expressions are the same. We do not discuss their semantics, as this has already been done in Secrion 2.1.3. The changes we made to While to form Goto are inspired by de Bruin from his work in [3]. In essence, our language Goto and the language used by de Bruin in [3] are syntactically the same.

Definition 3.1.1 (Syntax of Goto)**.** The definition of the syntax of Goto can be found in Table 3.1.

Num	$n ::= \texttt{0} \mid \texttt{1} \mid n\,\texttt{0} \mid n\,\texttt{1}$
Bexp	$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$
Aexp	$a ::= x \mid n \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ * } a_2 \mid a_1 \texttt{ - } a_2$
$\textbf{Stm}_{\text{G}}$	$S ::= x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$
	$\mid \texttt{goto } n$
Prog	$P ::= n{:}S \mid n{:}S \texttt{ \& } P$

Table 3.1: Syntax of the language Goto

Note. The syntax of both While and Goto do not include parenthesis, but they can be used to clarify how statements are composed on the meta-level. In general, the statement composition is considered to be right-associative, thus $S_1 \,;\, S_2 \,;\, S_3$ should be read as $S_1 \,;\, (S_2 \,;\, S_3)$.

The same holds for programs: $n_1 : S_1 \;\&\; n_2 : S_2 \;\&\; n_3 : S_3$ should be read as $n_1 : S_1 \;\&\; (n_2 : S_2 \;\&\; n_3 : S_3)$.

Note. The `else` parts of the `if` statements are considered to bind stronger than the statement composition.
`if` b `then` S_1 `else` $S_2 \,;\, S_3$ and (`if` b `then` S_1 `else` S_2) $;\, S_3$ are equal.

Note. Like with While, we extend the syntax for numerals with decimal digits, for the sake of readability.

The syntax category for programs **Prog** has been added to facilitate labels the goto statements can refer to, which is also done by de Bruin [3]. Labels in Goto are numerals and have a corresponding statement. The meta-variable P is introduced and ranges over **Prog**. Programs are concatenated with the & symbol, instead of the ; symbol, to show the different levels of composition more clearly.

Furthermore, the `while` b `do` S statement is replaced with the `goto` n statement.

3.2 Definitions and abbreviations

All programs in Goto are of the form $P := n_1 : S_1 \;\&\; \ldots \;\&\; n_k : S_k$. We abbreviate this by writing $P := [n_i : S_i]_{i=1}^{k}$.

The states we use for the semantics of Goto are defined the same way as for While in Section 2.1.2. They will be abbreviated for Goto in the same way they are for While:

$$\begin{aligned} s_a &= s[x \mapsto a] \\ s_{a,b} &= s[x \mapsto a][y \mapsto b] \\ s_{a,b,c} &= s[x \mapsto a][y \mapsto b][z \mapsto c] \end{aligned}$$

Where a, b, and c are integers. In case the value of some variable is unknown, we use the $\perp$ symbol.

Definition 3.2.1 (Well-formed programs)**.** A program $P := [n_i : S_i]_{i=1}^{k}$ from the syntax of Goto is *well-formed*, if and only if $\mathcal{N}[\![n_1]\!] = \mathbf{1}$ and for all $1 \leq i < k : \mathcal{N}[\![n_{i+1}]\!] = \mathcal{N}[\![n_i]\!] + \mathbf{1}$.

Note. The first label of a well-formed program can be `1`, `01`, `001`, and so on. The only requirement for a program to be well-formed is to have the first label being interpreted as the integer **1**, and consecutive labels being interpreted as consecutive integers.

Definition 3.2.2 (Called labels)**.** A label l (i.e. the integer) in program P from the syntax of Goto is *called*, if P contains a statement `goto` n with $\mathcal{N}[\![n]\!] = l$. This means that

1. l is called in $P := [n_i : S_i]_{i=1}^k$, if it is called in S_i for some $1 \leq i \leq k$.
2. For all statements S_1 and S_2, l is called in S_1 ; S_2, if it is called in S_1 or S_2.
3. For all statements S_1 and S_2, l is called in `if` b `then` S_1 `else` S_2, if it is called in S_1 or S_2.
4. Label l is called in `goto` n, if $\mathcal{N}[\![n]\!] = l$.

Definition 3.2.3 (Defined labels)**.** A label l in program $P := [n_i : S_i]_{i=1}^k$ from the syntax of Goto is *defined*, if $\mathcal{N}[\![n_i]\!] = l$ for some $1 \leq i \leq k$.

Definition 3.2.4 (Normal programs)**.** A program P from the syntax of Goto is *normal*, if all called labels in P are defined in P.

Note. For the rest of this thesis, we assume programs are normal and well-formed unless otherwise specified.

3.3 Semantic functions

The definitions of the semantic functions $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{B}$ for Goto are the same as for While and can be found in Tables 2.2, 2.3, and 2.4.

For proofs about certain properties of the semantics for Goto, we make use of the function $\mathcal{E}$ defined in 3.3.1. De Bruin defined such a function in [3] to be able to determine if the interpretation of statement S in state s terminates normally or via a goto statement.

If an operational interpretation (ie. with SOS or NS) of statement S in state s terminates without executing a goto statement, then $\mathcal{E}[\![S]\!]s = s'$, for some state s'.

If an operational interpretation of statement S in state s terminates after executing a `goto` n statement, then $\mathcal{E}[\![S]\!]s = \langle s', n \rangle$, for some state s'.

Definition 3.3.1 (Function $\mathcal{E}$)**.** We define the function $\mathcal{E}$ with type $\mathbf{Stm}_{\mathrm{G}} \rightarrow$

$\mathbf{State} \to \mathbf{State} \cup (\mathbf{State} \times \mathbf{Num})$ inductively by:

$$\mathcal{E}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\mathcal{E}[\![\texttt{skip}]\!]s = s$$

$$\mathcal{E}[\![S_1; S_2]\!]s = \begin{cases} \mathcal{E}[\![S_2]\!](\mathcal{E}[\![S_1]\!]s), & \text{if } \mathcal{E}[\![S_1]\!]s \in \mathbf{State} \\ \mathcal{E}[\![S_1]\!]s, & \text{if } \mathcal{E}[\![S_1]\!]s \in \mathbf{State} \times \mathbf{Num} \end{cases}$$

$$\mathcal{E}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} \mathcal{E}[\![S_1]\!]s, & \text{if } \mathcal{B}[\![b]\!]s = tt \\ \mathcal{E}[\![S_2]\!]s, & \text{if } \mathcal{B}[\![b]\!]s = ff \end{cases}$$

$$\mathcal{E}[\![\texttt{goto } n]\!]s = (s, n)$$

The function $\mathcal{E}$ essentially simulates structural operational semantics as defined in Chapter 4, until either the statement terminates, or a goto statement is encountered.

3.4 Design choices for Goto

3.4.1 Conditional statement

The language Goto is designed to resemble While. Therefore, it has the same structure as the language described by de Bruin in [3]. In particular, the conditional statement in Goto is `if` b `then` S_1 `else` S_2, which is a composite element of the syntax.

The non-modular nature of languages with goto statements makes it more difficult to define semantics for composite elements. The paper [9] by Saabas and Uustalu defines natural semantics for a language with goto. Their conditional statement has the form `if` b `then goto` n, which is notably *not* composite. In Chapter 5, we discuss the significance of this difference.

3.4.2 Labels

The labels in Goto are defined as numerals. This is not a necessity for structural operational or axiomatic semantics as defined by de Bruin in [3]. However, it helps define natural semantics, as shown by Saabas and Uustalu in [9]. The reason for this will become more clear in Chapter 5, where the rules for natural semantics for Goto are defined.

Chapter 4

Structural Operational Semantics

Structural Operational Semantics (SOS) follows the steps a machine would take when executing a program. The result is a sequence of steps. The derivation rules used in SOS result in a derivation tree for each step. However, the more important result of SOS is the sequence of steps, called the *derivation sequence*. Such a sequence is finite if and only if the program terminates or gets stuck (i.e. no rule can be applied, but no final state has been reached).

This chapter explains the thought process of constructing SOS rules for Goto, as done by de Bruin in [3].

4.1 SOS Rule design for Goto

The main difference between While and Goto is that While has while loops and Goto has labels and `goto` n statements.

However, this does not mean that altering the derivation rule for While results in correct semantics for the Goto Language. For example, one simple idea for a rule with `goto` n would be to skip all statements until the correct label is found. The problem is, that `goto` n may refer to a label at the start of the program. Skipping through the upcoming parts of the program is generally not sufficient as a rule for the `goto` n statement.

From this, it follows that there needs to be some construction to store the entire program, and in particular, the location of all labels and their corresponding statements, as shown by de Bruin in [3]. The new form for small-step transitions in Goto is compared to the transitions in While in Figure 4.1. P will not be altered in any rule and is only used for transitions with `goto` n statements. The rule for `goto` n statements will look up the appropriate statement in P and set S accordingly.

The SOS semantics for While are listed in Table 2.5. Our goal is to create

SOS transitions in While	SOS transitions in Goto
$\langle S, s\rangle \Rightarrow \langle S', s'\rangle$	$\langle P, S, s\rangle \Rightarrow \langle P, S', s'\rangle$
$\langle S, s\rangle \Rightarrow s'$	$\langle P, S, s\rangle \Rightarrow s'$

Figure 4.1: Comparison between structural operational semantic transitions in While and Goto

a set of SOS rules for Goto, which are closely related to the SOS rules of While. We will try to only replace the [while$_{\text{sos}}$] with a suitable rule for the goto statement at first.

4.1.1 The goto rule (SOS)

When the statement `goto` n is interpreted, the next statement should become the statement belonging to the label n (ignoring leading zeroes, see Definition *called labels* 3.2.2). But setting the next statement to be the statement corresponding to n when interpreting the statement `goto` n is not enough. A program would continue executing statements, even after the entire statement corresponding to label n has been executed.

Suppose we have $P := [n_i : S_i]_{i=1}^{5}$ and want to execute the statement `goto 2`. Then we should not just execute S_2 next, but $S_2; \ldots; S_5$. Thus the [goto$_{\text{sos}}$] should look like this:

$$\langle P,\ \texttt{goto}\ n,\ s\rangle \Rightarrow \langle P,\ S,\ s\rangle \quad [\text{goto}_{\text{sos}}]$$

$$\begin{aligned} \text{with } & P := [n_i : S_i]_{i=1}^{k}, \\ & \mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!] \text{ for some } 1 \le i \le k, \\ & \text{and } S := S_i; \ldots; S_k \end{aligned}$$

This rule can only be applied if P is normal (see Definition 3.2.4). If a statement `goto` n is encountered, but n is not defined in P, then the program is stuck.

While this may look like a solid rule, it does not yield the desired result yet. One example would be the following:

$$\begin{aligned} P := \quad & \texttt{1} : x := 0\ ;\ \texttt{goto 2}\ ;\ x := 1 \\ & \&\ \texttt{2} : \texttt{skip} \end{aligned} \tag{4.1}$$

The program P assigns the value 0 to x and then jumps over the next assignment so that the program terminates in a state s with $s\ x = 0$. Thus we expect a transition like $\langle P,\ (x := \texttt{0};\ \texttt{goto 2};\ x := \texttt{1});\ \texttt{skip},\ s_\perp\rangle \Rightarrow^* s_0$.

However, just replacing the [while$_{\text{sos}}$] rule with this [goto$_{\text{sos}}$] rule is not enough. Due to the composition rules, the "correct" interpretation is $\langle P,\ (x := \texttt{0};\ \texttt{goto 2};\ x := \texttt{1});\ \texttt{skip},\ s_\perp\rangle \Rightarrow^* s_1$. The rules used here are

$$
\begin{aligned}
&\langle P,\ (x := 0;\ \texttt{goto 2};\ x := 1);\ \texttt{skip},\ s_\perp\rangle \\
\Rightarrow\ &\langle P,\ (\texttt{goto 2};\ x := 1);\ \texttt{skip},\ s_0\rangle \\
\Rightarrow\ &\langle P,\ (\texttt{skip};\ x := 1);\ \texttt{skip},\ s_0\rangle \\
\Rightarrow\ &\langle P,\ x := 1;\ \texttt{skip},\ s_0\rangle \\
\Rightarrow\ &\langle P,\ \texttt{skip},\ s_1\rangle \\
\Rightarrow\ &s_1
\end{aligned}
$$

Figure 4.2: Derivation sequence for P as defined at 4.1, if only the [goto$_{\text{sos}}$] rule is added to the semantics

the [goto$_{\text{sos}}$] defined above, and the SOS rules for While defined in Table 2.5. The step-by-step transitions are listed in Figure 4.2, and a complete overview of the used rules can be found in Figure 4.3.

As stated before, the problem arises from the composition rules, specifically [comp$^1_{\text{sos}}$]. Given P as in Example 4.1, the desired transition is

$$\langle P,\ (\texttt{goto 2}\ ;\ x := 1)\ ;\ \texttt{skip},\ s_0\rangle \Rightarrow \langle P,\ \texttt{skip},\ s_0\rangle$$

Suppose we have the statement $S_1; S_2$. If a goto statement is executed in S_1, then the composition rule must make sure S_2 is *not* executed. But this should also extend to S_1 being composite with some goto statement.

4.1.2 Revised composition rules

The SOS of While has two composition rules [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$]. For any statement of the form $S_1; S_2$, [comp$^1_{\text{sos}}$] is applied if S_1 does not terminate in one step, while [comp$^2_{\text{sos}}$] is applied if S_1 does terminate in one step.

Instead of generalizing what S_1 can be, we can also have one rule for each option for statement S_1. Thus, we have one rule for $x := a; S$, another for $\texttt{skip}; S$, and so on, which is what de Bruin did in [3].

In the case of assignments, if statements, and skip statements, the new composition rules result in the same transition as before. If S_1 is a goto statement, then S_2 is ignored and the new statement is generated according to the label of the goto statement.

There also needs to be a rule for $(S_1; S_2)\ ;\ S_3$. Note that the statements $(S_1; S_2)\ ;\ S_3$ and $S_1; (S_2; S_3)$ should be equivalent, for all statements S_1, S_2, and S_3. Due to this, we can create a rule to shift the parenthesis to the right. This will be the associative-compositions rule [comp-assoc$_{\text{sos}}$].

Definition 4.1.1 (SOS for Goto)**.** The structural operational semantics for Goto are defined in Table 4.4.

$$\dfrac{\dfrac{\langle P,\ x := 0,\ s\rangle \Rightarrow s_0}{\langle P,\ x := 0\,;\ S,\ s\rangle \Rightarrow \langle P,\ S,\ s_0\rangle}\ [\mathrm{comp}^2_{\mathrm{sos}}]}{\langle P,\ (x := 0\,;\ S)\,;\ \texttt{skip},\ s\rangle \Rightarrow \langle P,\ S\,;\ \texttt{skip},\ s_0\rangle}\ [\mathrm{comp}^1_{\mathrm{sos}}]$$

$$\dfrac{\dfrac{\langle P,\ \texttt{goto 2},\ s_0\rangle \Rightarrow \langle P,\ \texttt{skip},\ s_0\rangle}{\langle P,\ \texttt{goto 2}\,;\ x := 1,\ s_0\rangle \Rightarrow \langle P,\ \texttt{skip}\,;\ x := 1,\ s_0\rangle}\ [\mathrm{comp}^1_{\mathrm{sos}}]}{\langle P,\ S\,;\ \texttt{skip},\ s_0\rangle \Rightarrow \langle P,\ (\texttt{skip}\,;\ x := 1)\,;\ \texttt{skip},\ s_0\rangle}\ [\mathrm{comp}^1_{\mathrm{sos}}]$$

$$\dfrac{\dfrac{\langle P,\ \texttt{skip},\ s_0\rangle \Rightarrow s_0}{\langle P,\ \texttt{skip}\,;\ x := 1,\ s_0\rangle \Rightarrow \langle P,\ x := 1,\ s_0\rangle}\ [\mathrm{comp}^2_{\mathrm{sos}}]}{\langle P,\ (\texttt{skip}\,;\ x := 1)\,;\ \texttt{skip},\ s_0\rangle \Rightarrow \langle P,\ x := 1\,;\ \texttt{skip},\ s_0\rangle}\ [\mathrm{comp}^1_{\mathrm{sos}}]$$

$$\dfrac{\langle P,\ x := 1,\ s_0\rangle \Rightarrow s_1}{\langle P,\ x := 1\,;\ \texttt{skip},\ s_0\rangle \Rightarrow \langle P,\ \texttt{skip},\ s_1\rangle}\ [\mathrm{comp}^2_{\mathrm{sos}}]$$

$$\langle P,\ \texttt{skip},\ s_1\rangle \Rightarrow s_1\ [\mathrm{skip}_{\mathrm{sos}}]$$

Figure 4.3: Derivation trees for each transition in figure 4.2, where $S := \texttt{goto 2}\,;\ x := 1$

$[\text{ass}_{\text{sos}}]$ $\quad \langle P,\ x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]$ $\quad \langle P,\ \texttt{skip}, s\rangle \Rightarrow s$

$[\text{if}^{\text{tt}}_{\text{sos}}]$ $\quad \langle P,\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle P,\ S_1,\ s\rangle$
if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$

$[\text{if}^{\text{ff}}_{\text{sos}}]$ $\quad \langle P,\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle P,\ S_2,\ s\rangle$
if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

$[\text{goto}_{\text{sos}}]$ $\quad \langle P,\ \texttt{goto}\ n,\ s\rangle \Rightarrow \langle P,\ S,\ s\rangle$
with $P := [n_i : S_i]_{i=1}^{k}$,
$\mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!]$ for some $1 \le i \le k$,
$S := S_i; \ldots; S_k$

$[\text{comp-assoc}_{\text{sos}}]$ $\quad \langle P,\ (S_1\ ;\ S_2)\ ;\ S_3,\ s\rangle \Rightarrow \langle P,\ S_1\ ;\ (S_2\ ;\ S_3),\ s\rangle$

$[\text{comp-ass}_{\text{sos}}]$ $\quad \langle P,\ x := a\ ;\ S,\ s\rangle \Rightarrow \langle P,\ S,\ s[x \mapsto a]\rangle$

$[\text{comp-skip}_{\text{sos}}]$ $\quad \langle P,\ \texttt{skip}\ ;\ S,\ s\rangle \Rightarrow \langle P,\ S,\ s\rangle$

$[\text{comp-if}_{\text{sos}}]$ $\quad \dfrac{\langle P,\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle P,\ S,\ s\rangle}{\langle P,\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ ;\ S_3,\ s\rangle \Rightarrow \langle P,\ S\ ;\ S_3,\ s\rangle}$

$[\text{comp-goto}_{\text{sos}}]$ $\quad \dfrac{\langle P,\ \texttt{goto}\ n,\ s\rangle \Rightarrow \langle P,\ S_2,\ s\rangle}{\langle P,\ \texttt{goto}\ n\ ;\ S_1,\ s\rangle \Rightarrow \langle P,\ S_2,\ s\rangle}$

Table 4.4: Structural operational semantics for Goto

Technically, the rules [comp-ass$_{\text{sos}}$] and [comp-skip$_{\text{sos}}$] are not needed, if one were to keep the [comp$^1_{\text{sos}}$] rule from While (with the addition of P in the configurations). This rule can only be applied to statements of the form $S_1; S_2$, if S_1 terminates in one single step. This is only the case for assignments and skip statements, thus [comp$^1_{\text{sos}}$] does not interfere with goto statements. However, it seemed more clear to have one rule each for every possible statement of S_1, even if the [comp-ass$_{\text{sos}}$] and [comp-skip$_{\text{sos}}$] seem a bit redundant.

These rules work on the statement level and not the program level. One can change the configurations from $\langle P, S, s\rangle$ to $\langle P, P', s\rangle$ and ignore the labels in P' for each rule. In this case, the starting configurations for a program P should be $\langle P, P, s\rangle$.

Another option is to add a start rule with the following transition:

$$\langle P, s\rangle \Rightarrow \langle P, S_1; \ldots S_k, s\rangle$$

This makes the starting configuration more explicit. However, in this thesis, we only use the rules as defined in Table 4.4.

4.2 Examples

Here are some example programs in Goto and their derivation sequences for certain states s, using the SOS rules found in Table 4.4.

4.2.1 Example 1

This is the same program used to demonstrate the need for different composition rules (Example 4.1).

$$\begin{aligned} P := \quad & \texttt{1} : x := 0 \,;\, \texttt{goto 2} \,;\, x := 1 \\ & \;\&\; \texttt{2} : \texttt{skip} \end{aligned}$$

With an arbitrary state s, the derivation sequence starting in

$$\langle P,\ (x := \texttt{0} \,;\, \texttt{goto 2} \,;\, x := \texttt{1}) \,;\, \texttt{skip},\ s\rangle$$

can be found in Figure 4.5. The parentheses have been added to show the effects of the [comp-assoc$_{\text{sos}}$] rule more clearly.

4.2.2 Example 2

This program computes the faculty of x, with the assumption that $x \geq 0$. We also define statements $S_1, S_{1,t}, S_2, S_{2,t}$ and S_3 to make the derivation sequence more readable.

$$
\begin{array}{ll}
\quad \langle P,\ (x := 0\ ;\ (\texttt{goto 2}\ ;\ x := 1))\ ;\ \texttt{skip},\ s_\perp \rangle & [\text{comp-assoc}_{\text{sos}}] \\
\Rightarrow \langle P,\ x := 0\ ;\ ((\texttt{goto 2}\ ;\ x := 1)\ ;\ \texttt{skip}),\ s_\perp \rangle & [\text{comp-ass}_{\text{sos}}] \\
\Rightarrow \langle P,\ (\texttt{goto 2}\ ;\ x := 1)\ ;\ \texttt{skip},\ s_0 \rangle & [\text{comp-assoc}_{\text{sos}}] \\
\Rightarrow \langle P,\ \texttt{goto 2}\ ;\ (x := 1\ ;\ \texttt{skip}),\ s_0 \rangle & [\text{comp-goto}_{\text{sos}}] \\
\Rightarrow \langle P,\ \texttt{skip},\ s_0 \rangle & [\text{skip}_{\text{sos}}] \\
\Rightarrow s_0 &
\end{array}
$$

Figure 4.5: Derivation sequence for Example 4.2.1

$$
\begin{array}{l|l}
P := \quad \texttt{1 : if } x \leq \texttt{1 then} & S_1 := \texttt{if } x \leq \texttt{1 then} \\
\qquad\qquad x := \texttt{1}; & \qquad\qquad S_{1,t} \\
\qquad\qquad \texttt{goto 3} & \qquad\quad \texttt{else} \\
\qquad\quad \texttt{else} & \qquad\qquad y := x\texttt{-1} \\
\qquad\qquad y := x\texttt{-1} & \hline
\qquad\quad \& \texttt{ 2 : if } \neg(y\texttt{=1}) \texttt{ then} & S_{1,t} := x := \texttt{1 ; goto 3} \\
\qquad\qquad x := x\texttt{*}y; & \hline
\qquad\qquad y := y\texttt{-1}; & S_2 := \texttt{if } \neg(y\texttt{=1}) \texttt{ then} \\
\qquad\qquad \texttt{goto 2} & \qquad\qquad S_{2,t} \\
\qquad\quad \texttt{else} & \qquad\quad \texttt{else} \\
\qquad\qquad \texttt{skip} & \qquad\qquad \texttt{skip} \\
\qquad\quad \& \texttt{ 3 : skip} & \hline
 & S_{2,t} := x := x\texttt{*}y\ ;\ y := y\texttt{-1}\ ;\ \texttt{goto 2} \\
 & \hline
 & S_3 := \texttt{skip}
\end{array}
$$

With an arbitrary state s, where $s\ x = 0$ (thus $s = s_{0,\perp}$), the derivation sequence starting in $\langle P,\ S_1; S_2; S_3,\ s_{3,\perp} \rangle$ can be found in Figure 4.6.

$$
\begin{array}{ll}
\quad \langle P,\ \texttt{if } x \leq \texttt{1 then } S_{1,t} \texttt{ else } y := x\texttt{-1}\ ;\ S_2\ ;\ S_3,\ s_{0,\perp} \rangle & \\
\Rightarrow \langle P,\ (x := \texttt{1}\ ;\ \texttt{goto 3})\ ;\ S_2\ ;\ S_3,\ s_{0,\perp} \rangle & \\
\Rightarrow \langle P,\ x := \texttt{1}\ ;\ \texttt{goto 3}\ ;\ S_2\ ;\ S_3,\ s_{0,\perp} \rangle & \\
\Rightarrow \langle P,\ \texttt{goto 3}\ ;\ S_2\ ;\ S_3,\ s_{1,\perp} \rangle & \\
\Rightarrow \langle P,\ \texttt{skip},\ s_{1,\perp} \rangle & \\
\Rightarrow s_{1,\perp} &
\end{array}
$$

Figure 4.6: The derivation sequence for Example 4.2.2 with $s_{0,\perp}$ as the initial state

With an arbitrary state s where $s\ x = 3$ (thus $s = s_{3,\perp}$), the derivation sequence starting in $\langle P,\ S_1; S_2; S_3,\ s_{3,\perp}\rangle$ can be found in Figure 4.7.

$$
\begin{aligned}
&\langle P,\ \texttt{if}\ x \leq \texttt{1}\ \texttt{then}\ S_{1,t}\ \texttt{else}\ y := x\texttt{-1}\,;\ S_2\,;\ S_3,\ s_{3,\perp}\rangle \\
\Rightarrow\ &\langle P,\ y := x\texttt{-1}\,;\ \texttt{if}\ \neg(y\texttt{=1})\ \texttt{then}\ S_{2,t}\ \texttt{else skip}\,;\ S_3,\ s_{3,\perp}\rangle \\
\Rightarrow\ &\langle P,\ \texttt{if}\ \neg(y\texttt{=1})\ \texttt{then}\ S_{2,t}\ \texttt{else skip}\,;\ S_3,\ s_{3,2}\rangle \\
\Rightarrow\ &\langle P,\ (x := x\texttt{*}y\,;\ y := y\texttt{-1}\,;\ \texttt{goto 2})\,;\ S_3,\ s_{3,2}\rangle \\
\Rightarrow\ &\langle P,\ x := x\texttt{*}y\,;\ (y := y\texttt{-1}\,;\ \texttt{goto 2})\,;\ S_3,\ s_{3,2}\rangle \\
\Rightarrow\ &\langle P,\ (y := y\texttt{-1}\,;\ \texttt{goto 2})\,;\ S_3,\ s_{6,2}\rangle \\
\Rightarrow\ &\langle P,\ y := y\texttt{-1}\,;\ \texttt{goto 2}\,;\ S_3,\ s_{6,2}\rangle \\
\Rightarrow\ &\langle P,\ \texttt{goto 2}\,;\ S_3,\ s_{6,1}\rangle \\
\Rightarrow\ &\langle P,\ \texttt{if}\ \neg(y\texttt{=1})\ \texttt{then}\ S_{2,t}\ \texttt{else skip}\,;\ S_3,\ s_{6,1}\rangle \\
\Rightarrow\ &\langle P,\ \texttt{skip}\,;\ \texttt{skip},\ s_{6,1}\rangle \\
\Rightarrow\ &\langle P,\ \texttt{skip},\ s_{6,1}\rangle \\
\Rightarrow\ &s_{6,1}
\end{aligned}
$$

Figure 4.7: The derivation sequence for Example 4.2.2 with $s_{3,\perp}$ as the initial state

4.2.3 Example 3

This simple example demonstrates an infinite derivation sequence.

$$P := \texttt{1:goto 1}$$

Some steps of the derivation sequence can be found in Figure 4.8.

$$\begin{aligned}
&\langle \texttt{1:goto 1},\ \texttt{goto 1},\ s_\perp \rangle && [\text{goto}_{\text{sos}}] \\
\Rightarrow\ &\langle \texttt{1:goto 1},\ \texttt{goto 1},\ s_\perp \rangle && [\text{goto}_{\text{sos}}] \\
\Rightarrow\ &\langle \texttt{1:goto 1},\ \texttt{goto 1},\ s_\perp \rangle && [\text{goto}_{\text{sos}}] \\
\Rightarrow\ &\ldots
\end{aligned}$$

Figure 4.8: The first steps of the derivation sequence for Example 4.2.3

The derivation sequence in Figure 4.8 repeats the same configuration. As a result, it will never terminate, as it will repeat the same steps.

More complex programs can also be proven to never terminate, by revealing a repeating pattern in the derivation sequence and proving that the program never exits this loop. Of course, non-termination of any program is in general unprovable [2].

4.2.4 Example 4

This is a short example of a program where its derivation sequence terminates in a stuck configuration. This cannot happen for normal programs, because the [goto$_{\text{sos}}$] and [comp-goto$_{\text{sos}}$] rules can always be applied to `goto` n and `goto` $n; S$, respectively.

$$P := \texttt{1:skip ; goto 2}$$

The derivation sequence can be found in Figure 4.9

$$\begin{aligned}
&\langle \texttt{1:skip ; goto 2},\ \texttt{skip ; goto 2},\ s_\perp \rangle && [\text{skip}_{\text{sos}}] \\
\Rightarrow\ &\langle \texttt{1:skip ; goto 2},\ \texttt{goto 2},\ s_\perp \rangle
\end{aligned}$$

Figure 4.9: The derivation sequence for Example 4.2.4

4.3 Properties

Theorem 4.3.1 (SOS is deterministic for well-formed normal programs in Goto)**.** *For all well-formed normal programs P, statements S, states s, and*

configurations γ and γ':
if $\langle P, S, s\rangle \Rightarrow \gamma$ and $\langle P, S, s\rangle \Rightarrow \gamma'$, then $\gamma = \gamma'$.

Proof. Let P, S, s, and γ, such that $\langle P, S, s\rangle \Rightarrow \gamma$ holds. We prove that if $\langle P, S, s\rangle \Rightarrow \gamma'$ holds, then $\gamma = \gamma'$ with induction on the shape of the derivation.

For the base cases, the derivations are of the form $\langle P, S, s\rangle \Rightarrow s'$:

Case $x := a$: The only rule which can be used to get $\langle P, x := a, s\rangle \Rightarrow s'$ is $[\text{ass}_{\text{sos}}]$, thus $s' = s[x \mapsto \mathcal{A}[\![a]\!]s]$.
As the same is true for $\langle P, x := a, s\rangle \Rightarrow s''$, we know that $s'' = s[x \mapsto \mathcal{A}[\![a]\!]s]$, thus $s' = s''$.

Case `skip`: Analogous to the case $x := a$.

Now, we prove the derivations of the form $\langle P, S, s\rangle \Rightarrow \langle P, S', s'\rangle$ deterministic:

Case `if` b `then` S_1 `else` S_2:

- Suppose the $[\text{if}^{\text{tt}}_{\text{sos}}]$ rule was applied: $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle P, S, s'\rangle$.
  Then we know that $S = S_1$, $s' = s$, and $\mathcal{B}[\![b]\!] = \mathbf{tt}$.

  Suppose the $[\text{if}^{\text{tt}}_{\text{sos}}]$ rule was applied: $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle P, S', s''\rangle$.
  Then we know that $S' = S_1$, $s'' = s$, and $\mathcal{B}[\![b]\!] = \mathbf{tt}$. Thus $\langle P, S, s'\rangle = \langle P, S', s''\rangle$.

  Suppose the $[\text{if}^{\text{ff}}_{\text{sos}}]$ rule was applied: $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle P, S', s''\rangle$.
  Then we know that $S' = S_2$, $s'' = s$, and $\mathcal{B}[\![b]\!] = \mathbf{ff}$.
  But $\mathcal{B}[\![b]\!] = \mathbf{ff}$ contradicts the assumption that $\mathcal{B}[\![b]\!] = \mathbf{tt}$ from applying the $[\text{if}^{\text{tt}}_{\text{sos}}]$ rule to get $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle P, S, s'\rangle$.
  As a result, $[\text{if}^{\text{ff}}_{\text{sos}}]$ cannot have been applied.

- The proof for $\mathcal{B}[\![b]\!] = \mathbf{ff}$ is analogous to the case of $\mathcal{B}[\![b]\!] = \mathbf{tt}$.

Case `goto` n:

- Suppose n is defined in P.
  The only rule which can be used to get $\langle P, \texttt{goto}\ n, s\rangle \Rightarrow \langle P, S, s'\rangle$ is $[\text{goto}_{\text{sos}}]$, thus with $P := [n_i : S_i]_{i=1}^{k}$, $\mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!]$ for some $1 \leq i \leq k$, we know that $S = S_i; \ldots; S_k$, and $s' = s$.

  If we have $\langle P, \texttt{goto}\ n, s\rangle \Rightarrow \langle P, S', s''\rangle$, then the label n must be defined in P, and the $[\text{goto}_{\text{sos}}]$ rule must have been applied.

As a result, with $P := [n_i : S_i]_{i=1}^k$, $\mathcal{N}[\![n]\!] = \mathcal{N}[\![n_j]\!]$ for some $1 \leq j \leq k$, and $S' = S_j; \ldots; S_k$, and $s'' = s$.
If $i = j$, then $\langle P, S, s' \rangle = \langle P, S', s'' \rangle$.
If $i \neq j$, then the premise of P being well-formed is not met.

- Suppose n is *not* defined in P.
  The premise of P being normal is not met, thus the property "if $\langle P, S, s \rangle \Rightarrow \gamma$ and $\langle P, S, s \rangle \Rightarrow \gamma'$, then $\gamma = \gamma'$" holds vacuously.

Case $S_1; S_2$: We split up the cases for S_1:

- **Case** $(S_1'; S_1''); S_2$: Analogous to the case $x := a$.
- **Case** $x := a; S_2$: Analogous to the case $x := a$.
- **Case** `skip`$; S_2$: Analogous to the case $x := a$.
- **Case** `if` b `then` S_1' `else` S_1'' $; S_2$:
  The only rule which can be used to get $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1''; S_2, s \rangle \Rightarrow \gamma$ is [comp-if$_{\text{sos}}$].
  From this, it follows that $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1'', s \rangle \Rightarrow \langle P, S, s \rangle$ for some S.
  Thus $\gamma = \langle P, S\ S_2, s \rangle$.

  If we have $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1''\ ;\ S_2, s \rangle \Rightarrow \gamma'$, then the [comp-if$_{\text{sos}}$] rule must have been applied.
  From this, it follows that $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1'', s \rangle \Rightarrow \langle P, S', s \rangle$ for some S'.
  Thus $\gamma = \langle P, S'\ S_2, s \rangle$.
  We already know that if $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1'', s \rangle \Rightarrow \langle P, S, s \rangle$ and $\langle P, \texttt{if}\ b\ \texttt{then}\ S_1'\ \texttt{else}\ S_1'', s \rangle \Rightarrow \langle P, S', s \rangle$, then $\langle P, S, s \rangle = \langle P, S', s \rangle$.
  As a result, $\gamma = \gamma'$.
- **Case** `goto` $n; S_2$: Analogous to the cases `if` b `then` S_1' `else` S_1'' $; S_2$ in combination with the case `goto` n.

□

Corollary. *For all well-formed normal programs P, statements S, states s, s' and s'':*
if $\langle P,\ S,\ s \rangle \Rightarrow^* s'$ and $\langle P,\ S,\ s \rangle \Rightarrow^* s''$, then $s' = s''$.

Definition 4.3.1 (Semantic equivalence)**.** For all well-formed normal programs $P_1 := [n_{1,i} : S_{1,i}]_{i=1}^k$ and $P_2 := [n_{2,i} : S_{2,i}]_{i=1}^q$ with $S_1 := S_{1,1}; \ldots; S_{1,k}$ and $S_2 := S_{2,1}; \ldots; S_{2,q}$, we say that P_1 and P_2 are *semantically equivalent* (denoted with $P_1 \sim P_2$) if the following predicate holds:

For all terminal or stuck configurations γ: $\langle P_1, S_1, s \rangle \Rightarrow^* \gamma$, if and only if $\langle P_2, S_2, s \rangle \Rightarrow^* \gamma$.

Proposition 4.3.2. *All programs that do not terminate are equivalent.*

Proof.

1. Let $P_1 := [n_{1,i} : S_{1,i}]_{i=1}^{k}$ and $P_2 := [n_{2,i} : S_{2,i}]_{i=1}^{q}$ be programs that do not terminate for any state s.
2. Let $S_1 := S_{1,1}; \ldots; S_{1,k}$ and $S_2 := S_{2,1}; \ldots; S_{2,q}$.
3. If $P_1 \sim P2$, then, for all states s and s': $\langle P_1, S_1, s\rangle \Rightarrow^* s'$, if and only if $\langle P_2, S_2, s\rangle \Rightarrow^* s'$.
4. From 1, it follows that for all states s and s': $\langle P_1, S_1, s\rangle \Rightarrow^* s'$ does not hold.
5. From 4, it follows that for all states s and s': $\langle P_1, S_1, s\rangle \Rightarrow^* s'$ only if $\langle P_2, S_2, s\rangle \Rightarrow^* s'$ holds.
6. From 1, it follows that there exist no states s and s' such that $\langle P_2, S_2, s\rangle \Rightarrow^* s'$ holds.
7. From 6, it follows that for all states s and s': $\langle P_1, S_1, s\rangle \Rightarrow^* s'$, if $\langle P_2, S_2, s\rangle \Rightarrow^* s'$ holds.
8. From 5 and 7, it follows that $\langle P_1, S_1, s\rangle \Rightarrow^* s'$, if and only if $\langle P_2, S_2, s\rangle \Rightarrow^* s'$ holds.

□

Theorem 4.3.3. *For all programs P statements S and states s and s': the derivation sequence $\langle P, S, s\rangle \Rightarrow^* s'$ exists and it does not contain the $[goto_{sos}]$ or $[comp\text{-}goto_{sos}]$ rules, if and only if $\mathcal{E}[\![S]\!]s = s'$.*

Proof. We prove this by induction on the shape of S:

Case $S = x := a$: The derivation sequence starting in $\langle P, x := a, s\rangle$ must look like this:

$$\langle P, x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

Interpreting S with $\mathcal{E}$ can be done directly from the definition of $\mathcal{E}$ (Definition 3.3.1):

$$\mathcal{E}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$$

Thus the theorem holds if $S = x := a$.

Case $S =$ `skip`: Analogous to $S = x := a$.

Case $S =$ `goto` n:

1. Proof for the "if" direction:
   The premise '$\mathcal{E}[\![\texttt{goto } n]\!]s = s'$' is never met, since by definition the result of $\mathcal{E}[\![\texttt{goto } n]\!]s$ has type $\mathbf{State} \times \mathbf{Num}$, and not type $\mathbf{State}$. As a result, the theorem holds for the "if" direction.

2. Proof for the "only if" direction:
   The premise '$\langle P, \texttt{goto } n, s\rangle \Rightarrow^* s'$ exists and it does not contain the [goto$_{\text{sos}}$] or [comp-goto$_{\text{sos}}$] rules' is never met, since the only the [goto$_{\text{sos}}$] rule can be applied to $\langle P, \texttt{goto } n, s\rangle$. As a result, the theorem holds for the "only if" direction.

Case $S = S_1; S_2$: For the induction hypothesis, we can assume that the theorem holds for S_1 and S_2.

1. Proof for the "if" direction:
   We can assume that $\mathcal{E}[\![S]\!]s = s'$ holds.
   In other words $\mathcal{E}[\![S_2]\!](\mathcal{E}[\![S_1]\!]s) = s'$ holds, by the definition of $\mathcal{E}$.
   From this, it follows that there exists some state s'', for which $\mathcal{E}[\![S_1]\!]s = s''$ holds.
   Then, it also follows that $\mathcal{E}[\![S_2]\!]s'' = s'$ holds. With our induction hypothesis, from $\mathcal{E}[\![S_1]\!]s = s''$ follows that there exists a derivation sequence $\langle P, S_1, s\rangle \Rightarrow^* s''$, and from $\mathcal{E}[\![S_2]\!]s'' = s'$ follows that there exists a derivation sequence $\langle P, S_2, s''\rangle \Rightarrow^* s'$, which both do not contain the [goto$_{\text{sos}}$] or [comp-goto$_{\text{sos}}$] rules.
   This means that the derivation sequence $\langle P, S_1; S_2, s\rangle \Rightarrow^* \langle P, S_2, s''\rangle \Rightarrow^* s'$ exists, and in particular $\langle P, S_1; S_2, s\rangle \Rightarrow^* s'$.

2. Proof for the "only if" direction:
   We can assume that the derivation sequence $\langle P, S_1; S_2, s\rangle \Rightarrow^* s'$ exists without the [goto$_{\text{sos}}$] or [comp-goto$_{\text{sos}}$] rules being applied.
   We can assume that a state s'' exists, such that $\langle P, S_1, s\rangle \Rightarrow^* s''$ holds. Otherwise, S_1 would never terminate when executed in state s, which would mean that the derivation sequence $\langle P, S_1; S_2, s\rangle \Rightarrow^* s'$ cannot exist.
   This means the derivation sequence $\langle P, S_1; S_2, s\rangle \Rightarrow^* s'$ looks like this: $\langle P, S_1; S_2, s\rangle \Rightarrow^* \langle P, S_2, s''\rangle \Rightarrow^* s'$.
   Since the [goto$_{\text{sos}}$] and [comp-goto$_{\text{sos}}$] rules are never applied in this derivation sequence, these rules are also not applied in the derivation sequence $\langle P, S_1, s\rangle \Rightarrow^* s''$.
   Because of our induction hypothesis, we know that from $\langle P, S_1, s\rangle \Rightarrow^* s''$ follows that $\mathcal{E}[\![S_1]\!]s = s''$, and from $\langle P, S_2, s''\rangle \Rightarrow^* s'$ follows that $\mathcal{E}[\![S_2]\!]s'' = s'$.
   By definition of $\mathcal{E}$, we know that $\mathcal{E}[\![S_1; S_2]\!] = \mathcal{E}[\![S_2]\!](\mathcal{E}[\![S_1]\!]s) = s'$.

Case $S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$: Analogous to $S = S_1; S_2$ □

Conjecture 4.3.1. For all programs P statements S and states s and s': the derivation sequence $\langle P, S, s\rangle \Rightarrow^* \langle P, S', s'\rangle$ does not contain either of the [goto$_{\text{sos}}$] or [comp-goto$_{\text{sos}}$] rules and the next rule is either the [goto$_{\text{sos}}$] or [comp-goto$_{\text{sos}}$], if and only if there exists a numeral n for which $\mathcal{E}[\![S]\!]s = (s', n)$ holds.

The Conjecture 4.3.1, states that for all statements S, the function $\mathcal{E}$ and SOS are equivalent up to the first goto statement. If during the execution of S no goto statement is encountered, then the resulting state from $\mathcal{E}$ is the final state of SOS.

Chapter 5

Natural Semantics

Reasoning about low-level languages (i.e. languages with goto statements) is believed to be difficult. This is due to the non-compositional nature that arises from control flow features, like goto statements, as mentioned by Saabas and Uustalu in [9]. In particular, natural semantics (NS) for such languages are harder to define.

Unlike programs in While, pieces of code in Goto can have multiple entry and exit points. The entry points are the labels, and the exit points are the goto statements, as well as the typical exit point at the end of a statement.

However, these challenges have been solved before. The language Exc is an extension to While and includes exceptions. In essence, with the expression `begin` S_1 `handle` e : S_2 `end`, the statement S_1 has multiple exit points in the form of statements like `raise` e, as shown by Nielson and Nielson in [6].

Furthermore, the multiple entry points have already been addressed for SOS in Chapter 4, which is based on the work of de Bruin [3]. That being said, the approach used for SOS works well, because of the step-by-step interpretation inherent to SOS. For NS, we will make use of a different construction.

To facilitate the multiple entry points, we make use of a program counter $pc \in \mathbb{N}$. This program counter points to a label in $P := [n_i : S_i]_{i=1}^{k}$. This allows us to construct a rule, which interprets the statement S_{pc}, and then continues with a new program counter. This idea is based on the work of Saabas and Uustalu in [9], where essentially transitions of the form $\langle S, pc, s\rangle \to \langle pc', s'\rangle$ are used. However, in their goto language, the only compositional statement is $S_1; S_2$. Our language Goto also has the compositional statement `if` b `then` S_1 `else` S_2. In the work from Saabas and Uustalu in [9] every primitive statement has its own label, which means that the `goto` n statement only ever points to one primitive statement. In our language Goto, the `goto` n statement can point to a block of statements with multiple exit points, something Saabas and Uustalu avoided [9].

For NS to work with Goto, we introduce the meta-variable c, which rages over the set $C := \{\circ, \bullet\}$ of control flow symbols. The $\circ$ symbol means we continue interpreting as normal, while the $\bullet$ symbol is used when a goto statement has been encountered.

The resulting transitions used for Goto are compared to NS transitions from While in Figure 5.1.

NS transition in While	NS transitions in Goto
$\langle S, s\rangle \to s'$	$\langle P, pc, s\rangle \to s'$
	$\langle S, s\rangle \to \langle pc, c, s'\rangle$

Figure 5.1: Comparison between structural operational semantic transitions in While and Goto

5.1 NS transitions for Goto

The NS rules for Goto are split into two groups: one for programs, and one for statements.

The rules for programs are concerned with finding the statement corresponding to the label equal to pc.

Definition 5.1.1 (NS rules for programs)**.** The NS rules for programs in goto can be found in Table 5.2.

$$[\text{find}^{\circ}_{\text{ns}}] \quad \frac{\langle S_{pc},\ s\rangle \to \langle pc',\ \circ,\ s''\rangle \qquad \langle P,\ pc+1,\ s''\rangle \to s'}{\langle P,\ pc,\ s\rangle \to s'}$$
$$\text{with } P := [n_i : S_i]_{i=1}^{k}, \text{ if } 1 \le pc \le k$$

$$[\text{find}^{\bullet}_{\text{ns}}] \quad \frac{\langle S_{pc},\ s\rangle \to \langle pc',\ \bullet,\ s''\rangle \qquad \langle P,\ pc',\ s''\rangle \to s'}{\langle P,\ pc,\ s\rangle \to s'}$$
$$\text{with } P := [n_i : S_i]_{i=1}^{k}, \text{ if } 1 \le pc \le k$$

$$[\text{end}_{\text{ns}}] \quad \langle P,\ pc,\ s\rangle \to s$$
$$\text{with } P := [n_i : S_i]_{i=1}^{k}, \text{ and } pc \notin \{1, \ldots, k\}$$

Table 5.2: Natural semantics for programs in Goto

The $[\text{find}^{\circ}_{\text{ns}}]$ rule executes the statement belonging to pc completely, if that statement ends at the normal exit point (i.e. no goto statement has been encountered).

The $[\text{find}^{\bullet}_{\text{ns}}]$ rule executes the statement belonging to pc up to and including the first goto statement. The appropriate new program counter value is then used to execute the next part of the program.

In case the pc is not defined in the program P, then the $[\text{end}_{\text{ns}}]$ rule will make sure the tree does not grow infinitely. The definition of $[\text{end}_{\text{ns}}]$ assumes P to be well-formed.

Note. The reason why pc ranges over the natural numbers (although integers would also have been fine) is the $[\text{find}^{\circ}_{\text{ns}}]$ rule. We found it more elegant to increase the value of pc by one with $+1$, rather than defining a function that increases the value of a syntactical numeral by one. Apart from that, since numerals allow for leading zeroes, we negate the problem that a label might be `02`, while the pc is `2`. In this case, `02` $\neq$ `2`, thus no rule can be applied. We avoid this problem by matching the labels and program counters with numbers in the semantic domain.

Note. The program counter pc' in rule $[\text{find}^{\circ}_{\text{ns}}]$ is not used. This will be discussed in more detail after the rules for statements in Goto have been defined.

The rules for statements are similar to those for While.

Definition 5.1.2 (NS rules for statements)**.** The NS rules for statements in Goto can be found in Table 5.3.

$$[\text{ass}_{\text{ns}}] \quad \langle x := a,\ s\rangle \to \langle 0,\ \circ,\ s[x \mapsto \mathcal{A}[\![a]\!]s]\rangle$$

$$[\text{skip}_{\text{ns}}] \quad \langle \texttt{skip},\ s\rangle \to \langle 0,\ \circ,\ s\rangle$$

$$[\text{comp}^{\circ}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to \langle pc,\ \circ,\ s''\rangle \qquad \langle S_2,\ s''\rangle \to \langle pc',\ c,\ s'\rangle}{\langle S_1; S_2,\ s\rangle \to \langle pc',\ c,\ s'\rangle}$$

$$[\text{comp}^{\bullet}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to \langle pc,\ \bullet,\ s'\rangle}{\langle S_1; S_2,\ s\rangle \to \langle pc,\ \bullet,\ s'\rangle}$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to \langle pc,\ c,\ s'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to \langle pc,\ c,\ s'\rangle} \quad \text{if } \mathcal{B}[\![b]\!]s = tt$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S_2,\ s\rangle \to \langle pc,\ c,\ s'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to \langle pc,\ c,\ s'\rangle} \quad \text{if } \mathcal{B}[\![b]\!]s = ff$$

$$[\text{goto}_{\text{ns}}] \quad \langle \texttt{goto}\ n,\ s\rangle \to \langle \mathcal{N}[\![n]\!],\ \bullet,\ s\rangle$$

Table 5.3: Natural semantics for statements in Goto

The rules $[\text{ass}_{\text{ns}}]$ and $[\text{skip}_{\text{ns}}]$ result in a configuration with the program counter set to 0, and the control flow symbol being $\circ$. As a result, the $[\text{find}^{\circ}_{\text{ns}}]$

and $[\text{find}^{\bullet}_{\text{ns}}]$ rules could have been defined to match on the program counter being 0 or not, instead of the control flow symbols $\circ$ and $\bullet$, respectively. We chose to include the control flow symbols, as we felt that it would make the applications of $[\text{find}^{\circ}_{\text{ns}}]$ and $[\text{find}^{\bullet}_{\text{ns}}]$ more clear.

Furthermore, we chose for $[\text{ass}_{\text{ns}}]$ and $[\text{skip}_{\text{ns}}]$ to result in a specific program counter, rather than any number, as that would make the rules for statements non-deterministic, even if it would not affect the interpretation of programs.

5.2 Examples

5.2.1 Example 1

This is the same program used as the first example for SOS.

$$\begin{aligned} P := \quad & \texttt{1} : x := 0 \,;\, \texttt{goto 2} \,;\, x := 1 \\ & \;\&\; \texttt{2} : \texttt{skip} \end{aligned}$$

The derivation tree looks like this:

$$\dfrac{\dfrac{\langle x := \texttt{0},\ s\rangle \to \langle 0, \circ, s_0\rangle \ [\text{ass}_{\text{ns}}] \qquad T_1}{\langle x := \texttt{0};\ \texttt{goto 2};\ x := \texttt{1},\ s\rangle \to \langle 2, \bullet, s_0\rangle}\ [\text{comp}^{\circ}_{\text{ns}}] \qquad T_2}{\langle P,\ 1,\ s\rangle \to \langle 3,\ s_0\rangle}\ [\text{find}^{\bullet}_{\text{ns}}]$$

Where T_1 is defined as:

$$\dfrac{\langle \texttt{goto 2},\ s_0\rangle \to \langle 2, \bullet, s_0\rangle \ [\text{goto}_{\text{ns}}]}{\langle \texttt{goto 2};\ x := \texttt{1},\ s_0\rangle \to \langle 2, \bullet, s_0\rangle}\ [\text{comp}^{\bullet}_{\text{ns}}]$$

and T_2 is defined as:

$$\dfrac{\langle \texttt{skip},\ s_0\rangle \to \langle 0, \circ, s_0\rangle \ [\text{skip}_{\text{ns}}] \qquad \langle P,\ 3,\ s_0\rangle \to \langle 3,\ s_0\rangle \ [\text{end}_{\text{ns}}]}{\langle P,\ 2,\ s_0\rangle \to \langle 3,\ s'\rangle}\ [\text{find}^{\circ}_{\text{ns}}]$$

A full version of this derivation tree can be found in Appendix A.1.

5.2.2 Example 2

This program computes the faculty of x, with the assumption that $x \geq 0$. We also define statements $S_1, S_{1,t}, S_2, S_{2,t}$ and S_3 to make the derivation sequence more readable.

$P :=$ 1 : `if` $x \leq 1$ `then`	$S_1 :=$ `if` $x \leq 1$ `then`
$x := 1$;	$S_{1,t}$
`goto 3`	`else`
`else`	$y := x$`-1`
$y := x$`-1`	$S_{1,t} := x := 1$; `goto 3`
& 2 : `if` $\neg(y$`=1`$)$ `then`	$S_2 :=$ `if` $\neg(y$`=1`$)$ `then`
$x := x$`*`y;	$S_{2,t}$
$y := y$`-1`;	`else`
`goto 2`	`skip`
`else`	$S_{2,t} := x := x$`*`y ; $y := y$`-1` ; `goto 2`
`skip`	$S_3 :=$ `skip`
& 3 : `skip`	

With an arbitrary state s where $s\ x = 0$ (thus $s = s_{0,\perp}$), the derivation tree starting in $\langle P,\ 1,\ s_{0,\perp}\rangle$ looks like this:

$$\dfrac{\dfrac{\dfrac{T_1}{\langle S_1,\ s_{3,\perp}\rangle \to \langle 3,\ \bullet,\ s_{1,\perp}\rangle}\ [\text{if}^{\text{tt}}_{\text{ns}}] \qquad \dfrac{T_2 \qquad \langle P,\ 4,\ s_{1,\perp}\rangle \to s_{1,\perp}}{\langle P,\ 3,\ s_{0,\perp}\rangle \to s_{1,\perp}}\ [\text{find}^{\circ}_{\text{ns}}]}{\langle P,\ 1,\ s_{0,\perp}\rangle \to s_{1,\perp}}\ [\text{find}^{\bullet}_{\text{ns}}]}\ [\text{end}_{\text{ns}}]$$

Where T_1 is defined as:

$$\dfrac{\langle x := 1, s_{0,\perp}\rangle \to \langle 0, \circ, s_{1,\perp}\rangle\ [\text{ass}_{\text{ns}}] \qquad \langle \texttt{goto 3}, s_{1,\perp}\rangle \to \langle 3, \bullet, s_{1,\perp}\rangle\ [\text{goto}_{\text{ns}}]}{\langle x := 1 \texttt{ ; goto 3}, s_{3,2}\rangle \to \langle 3, \bullet, s_{1,\perp}\rangle}\ [\text{comp}^{\circ}_{\text{ns}}]$$

and T_2 is defined as:

$$\langle \texttt{skip}, s_{1,\perp}\rangle \to \langle 0, \circ, s_{1,\perp}\rangle\ [\text{skip}_{\text{ns}}]$$

A full version of this derivation tree can be found in Appendix A.2.

With an arbitrary state s where $s\ x = 3$ (thus $s = s_{3,\perp}$), the derivation tree starting in $\langle P,\ 1,\ s_{3,\perp}\rangle$ looks like this:

$$\dfrac{\dfrac{\dfrac{\langle y := x\texttt{-1},\ s_{3,\perp}\rangle \to \langle 0,\ \circ,\ s_{3,2}\rangle\ [\text{ass}_{\text{ns}}]}{\langle S_1,\ s_{3,\perp}\rangle \to \langle 0,\ \circ,\ s_{3,2}\rangle}\ [\text{if}^{\text{ff}}_{\text{ns}}] \qquad \dfrac{T_1 \qquad T_2}{\langle P,\ 2,\ s_{3,2}\rangle \to \langle 4,\ s_{6,1}\rangle}\ [\text{find}^{\bullet}_{\text{ns}}]}{\langle P,\ 1,\ s_{3,\perp}\rangle \to \langle 4,\ s_{6,1}\rangle}\ [\text{find}^{\circ}_{\text{ns}}]$$

Where T_1 is defined as:

$$\dfrac{\dfrac{\langle x := x\texttt{*}y, s_{3,2}\rangle \to \langle 0, \circ, s_{6,2}\rangle \;[\text{ass}_{\text{ns}}] \qquad T_1'}{\langle x := x\texttt{*}y;\ y := y\texttt{-1};\ \texttt{goto 2}, s_{3,2}\rangle \to \langle 2, \bullet, s_{6,1}\rangle}\;[\text{comp}^{\circ}_{\text{ns}}]}{\langle \texttt{if } \neg(y\texttt{=1}) \texttt{ then } S_{2,t} \texttt{ else skip}, s_{3,2}\rangle \to \langle 2, \bullet, s_{6,1}\rangle}\;[\text{if}^{\text{tt}}_{\text{ns}}]$$

and T_1' is defined as:

$$\dfrac{\langle y := y\texttt{-1}, s_{6,2}\rangle \to \langle 0, \circ, s_{6,1}\rangle \;[\text{ass}_{\text{ns}}] \qquad \langle \texttt{goto 2}, s_{6,1}\rangle \to \langle 2, \bullet, s_{6,1}\rangle\;[\text{goto}_{\text{ns}}]}{\langle y := y\texttt{-1};\ \texttt{goto 2}, s_{6,2}\rangle \to \langle 2, \bullet, s_{6,1}\rangle}\;[\text{comp}^{\circ}_{\text{ns}}]$$

and T_2 is defined as:

$$\dfrac{\dfrac{\dfrac{\langle \texttt{skip},\ s_{6,1}\rangle \to \langle 0,\ \circ,\ s_{6,1}\rangle\;[\text{skip}_{\text{ns}}]}{\langle S_2,\ s_{6,1}\rangle \to \langle 0,\ \circ,\ s_{6,1}\rangle}\;[\text{if}^{\text{ff}}_{\text{ns}}] \qquad T_2'}{\langle P,\ 2,\ s_{6,1}\rangle \to \langle 4,\ s_{6,1}\rangle}\;[\text{find}^{\circ}_{\text{ns}}]$$

and T_2' is defined as:

$$\dfrac{\langle \texttt{skip},\ s_{6,1}\rangle \to \langle 0,\ \circ,\ s_{6,1}\rangle \;[\text{skip}_{\text{ns}}] \qquad \langle P,\ 4,\ s_{6,1}\rangle \to \langle 4,\ s_{6,1}\rangle\;[\text{end}_{\text{ns}}]}{\langle P,\ 3,\ s_{6,1}\rangle \to \langle 4,\ s_{6,1}\rangle}\;[\text{find}^{\circ}_{\text{ns}}]$$

A full version of this derivation tree can be found in Appendix A.3.

5.2.3 Example 3

This program is a simple example of a non-terminating program. As a result, it would generate an infinite derivation tree, which do not exist.

$$P := \texttt{1: goto 1}$$

A derivation tree starting in $\langle P, 1, s\rangle$ would look like this:

$$\dfrac{\langle \texttt{goto 1}, s\rangle \to \langle 1, \bullet, s\rangle\;[\text{goto}_{\text{ns}}] \qquad \dfrac{T_1 \qquad T_2}{\langle P, 1, s\rangle \to s'}\;[\text{find}^{\bullet}_{\text{ns}}]}{\langle P, 1, s\rangle \to s'}\;[\text{find}^{\bullet}_{\text{ns}}]$$

However, this tree with $\langle P, 1, s\rangle \to s'$ as its conclusion has a tree with the same conclusion as one of its premises. As a result, there does not exist a state s' for which the program P would terminate.

5.2.4 Example 4

This program shows how non-normal programs are handled in NS.

$$P := \texttt{1: skip; goto 2}$$

The derivation tree for P looks like this:

$$\frac{T_1 \quad \langle P, 2, s\rangle \to s}{\langle P, 1, s\rangle \to s} \begin{matrix}[\text{end}_{\text{ns}}] \\ [\text{find}^{\bullet}_{\text{ns}}]\end{matrix}$$

With T_1 defined as:

$$\frac{\langle \texttt{skip}, s\rangle \to \langle 0, \circ, s\rangle \ [\text{skip}_{\text{ns}}] \qquad \langle \texttt{goto 2}, s\rangle \to \langle 2, \bullet, s\rangle \ [\text{goto}_{\text{ns}}]}{\langle \texttt{skip ; goto 2}, s\rangle \to \langle 2, \bullet, s\rangle} \ [\text{comp}^{\circ}_{\text{ns}}]$$

The $[\text{end}_{\text{ns}}]$ rule results in non-normal programs being terminated, if some `goto` n statement is encountered, with $\mathcal{N}[\![n]\!]$ not being a defined label.

5.3 Properties

Note. While this chapter has shown a way of defining natural semantics for a language with goto statements, this form of NS is not entirely compositional. For every $[\text{find}^{\circ}_{\text{ns}}]$ or $[\text{find}^{\bullet}_{\text{ns}}]$ rule that is applied, we only look at the part of the program corresponding to the label the program counter points to. However, then we consider the entire program again to find the next statement. In the work by Saabas and Uustalu [9], the NS is entirely compositional, meaning that reasoning about some program is done by reasoning about small individual parts, which are then combined with their composition rule.

Lemma 5.3.1 (NS is deterministic for statements in Goto)**.** *For all statements S, states s, s' and s'', natural numbers pc and pc', and control symbols c and c': if $\langle S, s\rangle \to \langle pc, c, s'\rangle$ and $\langle S, s\rangle \to \langle pc', c', s''\rangle$, then $pc = pc'$, $c = c'$, and $s' = s''$.*

Lemma 5.3.1 can be proven in a similar manner SOS was proven to be deterministic.

Theorem 5.3.2 (NS is deterministic for well-formed normal programs in Goto)**.** *For all well-formed normal programs P, natural numbers pc, and states s, s' and s'': if $\langle P,\ pc,\ s\rangle \to s'$ and $\langle P,\ pc,\ s\rangle \to s''$, then $s' = s''$.*

Proof. Proof by induction on the shape of the derivation tree.

We define the property $Q(T)$ for derivation trees T:

For all well-formed normal programs P, natural numbers pc, and states s, s' and s'':
if T has $\langle P,\ pc,\ s\rangle \to s'$ as its conclusion, then for all derivation trees T' with $\langle P,\ pc,\ s\rangle \to s''$ as its conclusion, $s' = s''$ holds.

Base case - axiom schemes:

1. Assume the last step in tree T is $[\text{end}_{\text{ns}}]$.
   Then we know that with $P := [n_i : S_i]_{i=1}^{k}$, $pc \notin \{1, \ldots, k\}$ and T looks like this:
   $$\langle P, pc, s\rangle \to s \ [\text{end}_{\text{ns}}]$$
   Thus $s' = s$.

   Suppose we have a derivation tree T' with $\langle P, pc, s\rangle \to s$ as its conclusion.
   The last step of T' cannot have been $[\text{find}^{\circ}_{\text{ns}}]$, because then $1 \leq pc \leq k$ must hold, which contradicts the fact $pc \notin \{1, \ldots, k\}$.
   The last step of T' cannot have been $[\text{find}^{\bullet}_{\text{ns}}]$ for the same reason.
   Thus, the last step must have been $[\text{end}_{\text{ns}}]$, which means T' looks like this:
   $$\langle P, pc, s\rangle \to s \ [\text{end}_{\text{ns}}]$$
   Thus $s'' = s$, from which $s' = s''$ follows.

   As a result, $P(T)$ holds for all derivation trees T with one rule application.

Induction hypothesis (IH) - composite rules:

We may assume that the property $P(T^*)$ holds for all of the premises T^* of the composite rules.

1. Assume the last step in tree T is $[\text{find}^{\circ}_{\text{ns}}]$.
   Then we know that with $P := [n_i : S_i]_{i=1}^{k}$, $1 \leq pc \leq k$, $S := S_{pc}$ and T looks like this:
   $$\dfrac{\dfrac{T_1^*}{\langle S,\ s\rangle \to \langle pc',\ \circ,\ s''\rangle}\ [\ldots] \qquad \dfrac{T_2^*}{\langle P,\ pc+1,\ s''\rangle \to s'}\ [\ldots]}{\langle P,\ pc,\ s\rangle \to s'}\ [\text{find}^{\circ}_{\text{ns}}]$$
   We know that T_1^* is deterministic (Theorem 5.3.1)

   Suppose the last rule in tree T' with $\langle P,\ pc,\ s\rangle \to s'''$ is $[\text{end}_{\text{ns}}]$.
   Then with $P := [n_i : S_i]_{i=1}^{k}$, pc must be greater than k, or smaller than 1.

But this contradicts the fact that $1 \leq pc \leq k$, thus $[\text{end}_{\text{ns}}]$ cannot have been the last rule of T'.

Suppose the last rule in tree T' with $\langle P,\ pc,\ s\rangle \to s'''$ is $[\text{find}^{\circ}_{\text{ns}}]$.
Then, due to the existence of T_1^*, T' must look like this:

$$\dfrac{\dfrac{T_1^*}{\langle S,\ s\rangle \to \langle pc',\ \circ,\ s''\rangle}\ [\dots] \qquad \dfrac{T_3^*}{\langle P,\ pc+1,\ s''\rangle \to s'''}\ [\dots]}{\langle P,\ pc,\ s\rangle \to s'''}\ [\text{find}^{\circ}_{\text{ns}}]$$

Since we know that $P(T_2^*)$ holds, it must be the case that $T_2^* = T_3^*$, since they have the same conclusion. As a result, we also know that $s' = s'''$.
Therefore, $P(T)$ holds, if the last step in tree T is $[\text{find}^{\circ}_{\text{ns}}]$.

Suppose the last rule in tree T' with $\langle P,\ pc,\ s\rangle \to s'''$ is $[\text{find}^{\bullet}_{\text{ns}}]$.
Then with $P := [n_i : S_i]_{i=1}^k$ and $S := S_{pc}$, we must have $\langle S,\ s\rangle \to \langle pc', \bullet, s'''\rangle$.
But we know from the fact that T_1^* exists, that the transition from $\langle S,\ s\rangle$ goes to $\langle pc', \circ, s''\rangle$.
This means that $[\text{find}^{\bullet}_{\text{ns}}]$ cannot have been the last rule of T'.

2. The proof for $[\text{find}^{\bullet}_{\text{ns}}]$ is analogous to the proof for $[\text{find}^{\circ}_{\text{ns}}]$.

□

Definition 5.3.1 (Semantic equivalence with NS)**.** For all well-formed normal programs P_1 and P_2, P_1 and P_2 are semantically equivalent (denoted with $P_1 \sim P_2$) if the following predicate holds:
For all states s and s': $\langle P_1, s\rangle \to s'$, if and only if $\langle P_2, s\rangle \to s'$.

Lemma 5.3.3 (SOS and NS result in equivalent interpretations for statements)**.** *Let $P := [n_i : S_i]_{i:=1}^k$ be a normal and well-formed program. For all natural numbers pc and i with $1 \leq pc \leq k$ and $1 \leq i \leq k$, and states s and s':*

1. *If $i < k$ and $\langle S_i, s\rangle \to \langle pc, \circ, s'\rangle$,*
   then $\langle P,\ S_i; \dots; S_k,\ s\rangle \Rightarrow^* \langle P,\ S_{i+1}; \dots; S_k,\ s'\rangle$

2. *If $\langle S_k, s\rangle \to \langle pc, \circ, s'\rangle$,*
   then $\langle P,\ S_k,\ s\rangle \Rightarrow^* s'$

3. *If $\langle S_i, s\rangle \to \langle pc, \bullet, s'\rangle$,*
   then $\langle P,\ S_i; \dots; S_k,\ s\rangle \Rightarrow^* \langle P,\ S_{pc}; \dots; S_k,\ s'\rangle$

Proof.

1. This can be proven by the fact that the [goto_{ns}] rule is *not* used in the derivation tree with $\langle S_i, s\rangle \to \langle pc, \circ, s'\rangle$ as its conclusion.
   As a result, the derivation sequence starting in $\langle P,\ S_i; \ldots; S_k,\ s\rangle$ does no make use of the [goto_{sos}] or [comp-goto_{sos}] rules. This means that $\langle P,\ S_i; \ldots; S_n,\ s\rangle \Rightarrow^* \langle P,\ S_{i+1}; \ldots; S_k,\ s'\rangle$.

2. This can be proven by the fact that the [goto_{ns}] rule is *not* used in the derivation tree with $\langle S_i, s\rangle \to \langle pc, \circ, s'\rangle$ as its conclusion.
   Since the program terminates, if the statement corresponding to the last label in P terminates without encountering a goto statement, the derivation sequence must end in a final configuration (i.e. a state).

3. This can be proven by the fact that the [goto_{ns}] rule *is* used in the derivation tree with $\langle S_i, s\rangle \to \langle pc, \bullet, s'\rangle$ as its conclusion.
   As a result, the derivation sequence starting in $\langle P,\ S_i; \ldots; S_k,\ s\rangle$ must make use of the [goto_{sos}] or [comp-goto_{sos}] rule, before the statement S_i has been executed completely. That goto statement must be `goto` n, with $\mathcal{N}[\![n]\!] = pc$, since otherwise the NS interpretation would have encountered a different goto statement.
   Since the goto statement jumps the program to the statement S_{pc}, the configuration $\langle P,\ S_{pc}; \ldots; S_k,\ s'\rangle$ must be reached after the first [goto_{sos}] or [comp-goto_{sos}] rule.

□

Conjecture 5.3.1 (SOS and NS result in equivalent interpretations)**.** For all programs $P = [n_i : S_i]_{i:=1}^k$, natural numbers i with $1 \le i \le k$, and states s and s': $\langle P,\ S_i; \ldots; S_k,\ s\rangle \Rightarrow^* s'$, if and only if $\langle P, i, s\rangle \to s'$.

Corollary. *For all programs $P = [n_i : S_i]_{i:=1}^k$, and states s and s':*
$\langle P,\ S_1; \ldots; S_k,\ s\rangle \Rightarrow^* s'$*, if and only if* $\langle P, 1, s\rangle \to s'$.

Chapter 6

Axiomatic Semantics

Axiomatic semantics (AS) is not about specific states like in SOS or NS, but it is about properties of the starting and (if it exists) final states.

The assertion

$$\{\, Q_1 \,\}\ P\ \{\, Q_2 \,\}$$

is called a Hoare triple, and it consists of two predicates, Q_1 and Q_2, as well as a program P. It reads "if Q_1 is true in the starting state and we execute program P, then Q_2 is true in the final state if the program terminates." The Predicates Q_1 and Q_2 are called the pre- and postcondition respectively. Note that AS does not prove that P terminates. Termination is generally undecidable [2], which is why we read the Hoare triples as described above. This results in Hoare triples being about partial correctness, which is why we index the AS rules with a lowercase p.

The AS rules for While can generally be carried over for Goto. The main concerns are about handling labels and, of course, a rule for the goto statement. For this, the Hoare triple will change a bit, which is described in the upcoming section. The semantics in this chapter are essentially the same as described by de Bruin in [3].

6.1 The goto rule (AS)

Since the goto statement replaces the while statement, it makes sense to analyze the [while$_\text{p}$] rule to understand how a goto rule can be constructed.

$$\frac{\{\mathcal{B}[\![b]\!] \wedge Q\}\ S\ \{Q\}}{\{Q\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\neg\mathcal{B}[\![b]\!] \wedge Q\}}\ [\text{while}_\text{p}]$$

Every time the body of the while loop is executed (i.e. the statement S), the assertion $\mathcal{B}[\![b]\!] \wedge Q$ must hold. Another way to look at this is if the program jumps back to execute S again, $\mathcal{B}[\![b]\!] \wedge Q$ must hold. For [while$_\text{p}$], this is the

case, since Q must hold after executing S, and $\mathcal{B}[\![b]\!]$ must hold if the loop is executed again.

For the goto statement, we can make use of the idea that certain predicates should hold if the program jumps to a different point of the program. Since goto statements can be anywhere in any program, we need a way to keep track of all relevant preconditions throughout the axiomatic interpretation. To see which preconditions are relevant, consider the following inference tree for While:

$$\dfrac{\dfrac{\dots}{\{Q_1\}\ S_1\ \{Q_2\}} \qquad \dfrac{\dfrac{\dots}{\{Q_2\}\ S_2\ \{Q_3\}} \qquad \dfrac{\dfrac{\dots}{\{Q_3\}\ S_3\ \{Q_4\}} \qquad \dfrac{\dots}{\{Q_4\}\ S_4\ \{Q_5\}}}{\{Q_1\}\ S_3; S_4\ \{Q_5\}}}{\{Q_1\}\ S_2; S_3; S_4\ \{Q_5\}}}{\{Q_1\}\ S_1; S_2; S_3; S_4\ \{Q_5\}}$$

We see that every statement S_i has a corresponding precondition Q_i and postcondition Q_{i+1}. We can rewrite the inference tree above like this:

$$\dfrac{\dfrac{\dots}{\{Q_1\}\ S_1\ \{Q_2\}} \qquad \dfrac{\dots}{\{Q_2\}\ S_2\ \{Q_3\}} \qquad \dfrac{\dots}{\{Q_3\}\ S_3\ \{Q_4\}} \qquad \dfrac{\dots}{\{Q_4\}\ S_4\ \{Q_5\}}}{\{Q_1\}\ S_1; S_2; S_3; S_4\ \{Q_5\}}$$

While this inference tree does not follow from the AS rules described by Nielson and Nielson in [6], we can use it to show why the AS ideas from de Bruin in [3] work.

Suppose we have a program $P = n_1 : S_1\ \&\ n_2 : S_2\ \&\ n_3 : S_3\ \&\ n_4 : S_4$. If we have an inference tree like the one above, then for every statement `goto` n_i, we need to make sure that the predicate Q_i holds when executing the goto statement. In particular, we need to keep track of the preconditions of every statement which corresponds to a label.

For this, in [3], de Bruin introduces a dictionary D:

$$D ::= n : Q \mid n : Q,\ D$$

This dictionary is a list of label-predicate pairs.

For programs P, we have introduced the notation $P = [n_i : S_i]_{i=1}^k$ as a shorthand for $P = n_1 : S_1\ \&\ \dots\ \&\ n_k : S_k$. We will do the same for dictionaries, where $D = [n_i : Q_i]_{i=1}^k$ is short for $D = n_1 : Q_1,\ \dots\ ,\ n_k : Q_k$.

The Hoare triples need to be adjusted to include this dictionary. We make two kinds of Hoare triples: one for programs, and one for statements. For statements, we include the dictionary the same way as de Bruin did in [3]:

$$\{Q_1\}\ S\ \{Q_2\} \text{ becomes } \langle D \mid \{Q_1\}\ S\ \{Q_2\}\rangle$$

For programs, we keep the original notation for Hoare triples without a dictionary. The reasons for this are explained in section 6.2.

Now, we can construct a rule for goto statements like this [3]:

$$\begin{array}{l}\langle D \mid \{Q\}\ \texttt{goto}\ n\ \{\ ?\ \}\rangle \quad [\text{goto}_\text{p}] \\ \text{with } D := [n_i : Q_i]_{i=1}^k, \\ \mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!] \text{ and } Q = Q_i \text{ for some } 1 \leq i \leq k\end{array}$$

This rule forces the precondition of any `goto` n statement to be the same precondition for $S_{n'}$, if $\mathcal{N}[\![n]\!] = \mathcal{N}[\![n']\!]$. As a result, before a program executes a statement like `goto` n, we know that the assertion Q holds, so if the program then jumps to the corresponding label, the precondition of the next statement holds.

The question now is what the postcondition should be. It can be tempting to set it to Q as well since that assertion should hold after executing a goto statement. However, normal programs in Goto (Definition 3.2.4), always have a statement, which is executed after some goto statement has been executed. In particular, a program does not terminate immediately after the execution of a goto statement. As a result, the postcondition for the goto statement has no real meaning. Postconditions should hold if the statement is executed and terminates after the precondition holds.

As a result, the postcondition can be set to false. Since false implies everything, the consequence rule can be used to change the postcondition to the precondition of the statement directly following the goto statement.

$$\begin{array}{l}\langle D \mid \{Q\}\ \texttt{goto}\ n\ \{\ \text{false}\ \}\rangle \quad [\text{goto}_\text{p}] \\ \text{with } D := [n_i : Q_i]_{i=1}^k, \\ \mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!] \text{ and } Q = Q_i \text{ for some } 1 \leq i \leq k\end{array}$$

Consider the inference tree from before again (we exclude the dictionary for simplicity for now):

$$\dfrac{\dfrac{\dots}{\{Q_1\}\ S_1\ \{Q_2\}} \quad \dfrac{\dots}{\{Q_2\}\ S_2\ \{Q_3\}} \quad \dfrac{\dots}{\{Q_3\}\ S_3\ \{Q_4\}} \quad \dfrac{\dots}{\{Q_4\}\ S_4\ \{Q_5\}}}{\{Q_1\}\ S_1; S_2; S_3; S_4\ \{Q_5\}}$$

Suppose that the last statement in S_1 is `goto` 3. Then the postcondition of S_1, namely Q_2, would be false. However, then the precondition of S_2 would be false as well. This means that the Hoare triple of S_2 has lost meaning because it reads "if the precondition Q_2 holds, ...", but the precondition never holds. As a result, one could set Q_3 to false. However, the precondition of `goto` 3 must equal Q_3, which will in general not be false. This is solved with the consequence rule like this:

$$\dfrac{\dfrac{\dfrac{\dots}{\{Q_1\}\ S_1\ \{\text{false}\}}}{\{Q_1\}\ S_1\ \{Q_2\}} \quad \dfrac{\dots}{\{Q_2\}\ S_2\ \{Q_3\}} \quad \dfrac{\dots}{\{Q_3\}\ S_3\ \{Q_4\}} \quad \dfrac{\dots}{\{Q_4\}\ S_4\ \{Q_5\}}}{\{Q_1\}\ S_1; S_2; S_3; S_4\ \{Q_5\}}$$

Now, with the last statement of S_1 being `goto` 3, the precondition of `goto` 3 does not need to result in the precondition of S_2, because the postcondition of `goto` 3 is false, and false implies Q_2.

If the postcondition of `goto` 3 would be Q_3 (i.e. the same as its precondition), then we need to show that Q_3 implies Q_2, which might not be possible depending on the program.

6.2 Labels

For Goto, we can use the same composition rule for statements, as used for While (after adding the dictionary to it). However, there also needs to be a rule for programs. First, let us see how the composition rule for statements works.

$$\frac{\langle D \mid \{Q_1\}\ S_1\ \{Q_2\}\rangle \quad \langle D \mid \{Q_2\}\ S_2\ \{Q_3\}\rangle}{\langle D \mid \{Q_1\}\ \ S_1\ ;\ S_2\ \ \{Q_3\}\rangle}\ [\text{comp}_\text{p}]$$

For any two consecutive statements S_1 and S_2, if assertion Q_1 holds, then after executing S_1 some intermediate assertion Q_2 holds. This postcondition of S_1 is simultaneously the precondition of S_2. Executing S_2 in a state where Q_2 holds results in the postcondition of S_2, which is Q_3. This is also the postcondition of $S_1\ ;\ S_2$.

Note here that if S_1 is a goto statement, Q_2 would be false. But if the consequence rule is applied on $\langle D \mid \{Q_1\}\ S_1\ \{Q_2\}\rangle$, Q_2 can be anything, thus in particular a useful assertion for the precondition of S_2.

Also, for any two consecutive statements, the postcondition of the first statement is the precondition of the second statement. This can be used to make a rule for programs in Goto.

One rule for programs in goto can look like this [3]:

$$\frac{\langle D \mid \{Q_1\}\ S_1\ \{Q_2\}\rangle \quad \dots \quad \langle D \mid \{Q_k\}\ S_k\ \{Q_{k+1}\}\rangle}{\{Q_1\}\ \ P\ \ \{Q_{k+1}\}}\ [\text{exp}_\text{p}]$$

Where $P := [n_i : S_i]_{i=1}^k$, and $D := [n_i : Q_i]_{i=1}^k$. This rule is called the expand rule.

Here, the postcondition of every statement corresponding to a label is also the precondition of the next label. Also, the preconditions of all statements corresponding to a label are stored in D. However, the initial precondition of the judgement $\{Q_1\}\ \ P\ \ \{Q_{k+1}\}$ might not be a suitable assertion for the dictionary. As a result, we also have a consequence rule for programs.

Definition 6.2.1 (AS rules for Goto)**.** The AS rules for statements and programs in Goto can be found in Table 6.1

$$[\text{ass}_\text{p}] \quad \langle D \mid \{Q[x \mapsto \mathcal{A}[\![a]\!]]\}\ x := a\ \{Q\}\rangle$$

$$[\text{skip}_\text{p}] \quad \langle D \mid \{Q\}\ \texttt{skip}\ \{Q\}\rangle$$

$$[\text{goto}_\text{p}] \quad \langle D \mid \{Q\}\ \texttt{goto}\ n\ \{\text{false}\}\rangle$$
with $D := [n_i : Q_i]_{i=1}^k$,
$\mathcal{N}[\![n]\!] = \mathcal{N}[\![n_i]\!]$ and $Q = Q_i$ for some $1 \leq i \leq k$

$$[\text{cons}_\text{p}^1] \quad \frac{\{Q_1'\}\ P\ \{Q_2'\}}{\{Q_1\}\ P\ \{Q_2\}}$$
if $Q_1 \Rightarrow Q_1'$ and $Q_2' \Rightarrow Q_2$

$$[\text{cons}_\text{p}^2] \quad \frac{\langle D \mid \{Q_1'\}\ S\ \{Q_2'\}\rangle}{\langle D \mid \{Q_1\}\ S\ \{Q_2\}\rangle}$$
if $Q_1 \Rightarrow Q_1'$ and $Q_2' \Rightarrow Q_2$

$$[\text{comp}_\text{p}] \quad \frac{\langle D \mid \{Q_1\}\ S_1\ \{Q_2\}\rangle \quad \langle D \mid \{Q_2\}\ S_2\ \{Q_3\}\rangle}{\langle D \mid \{Q_1\}\ \ S_1\ ;\ S_2\ \ \{Q_3\}\rangle}$$

$$[\text{if}_\text{p}] \quad \frac{\langle D \mid \{\mathcal{B}[\![b]\!] \wedge Q_1\}\ S_1\ \{Q_2\}\rangle \quad \langle D \mid \{\neg\mathcal{B}[\![b]\!] \wedge Q_1\}\ S_2\ \{Q_2\}\rangle}{\langle D \mid \{Q_1\}\ \texttt{if}\ \ b\ \texttt{then}\ \ S_1\ \texttt{else}\ \ S_2\ \{Q_2\}\rangle}$$

$$[\text{exp}_\text{p}] \quad \frac{\langle D \mid \{Q_1\}\ S_1\ \{Q_2\}\rangle \quad \ldots \quad \langle D \mid \{Q_k\}\ S_k\ \{Q_{k+1}\}\rangle}{\{Q_1\}\ P\ \{Q_{k+1}\}}$$
with $P := [n_i : S_i]_{i=1}^k$
and $D := [n_i : Q_i]_{i=1}^k$

Table 6.1: Axiomatic semantics for Goto

In other words for the case of assertions of the form $\langle D \mid \{Q_1\}\ S\ \{Q_2\}\rangle$, if statement S is executed in a state s in which Q_1 holds, then either:

- no goto statements are executed during the execution of S, which means if the program terminates in state s', then Q_2 holds in that state.
- or some goto statement is executed, say `goto` n. At this point, it does not matter if Q_1 or Q_2 holds in the current state. It only matters if the precondition corresponding to label n holds in the current state. For this, the dictionary D is used, as it includes some pair $n : Q$, and the predicate Q must hold at the time the statement `goto` n is encountered.

6.3 Examples

6.3.1 Example 1

This example is the same as in equation 4.1, which was used to show complications of the comp rule in SOS. We define P and S_1 as follows:

$$\begin{aligned} P := \quad & \texttt{1} : x := 0 \,;\, \texttt{goto 2} \,;\, x := 1 \\ & \;\&\; \texttt{2} : \texttt{skip} \end{aligned}$$

$$S_1 := \; x := 0 \,;\, \texttt{goto 2} \,;\, x := 1$$

Suppose we want to prove that if P terminates, then $x = 0$ holds for any precondition. This means that an inference tree with the following conclusion must exist:

$$\{ \text{ true } \}\ \texttt{1} : x := 0; \texttt{ goto 2}; \ x := 1 \ \&\ \texttt{2} : \texttt{skip}\ \{ \ x = 0 \ \}$$

This is one step-by-step example of creating such an inference tree:

First, apply the [exp_p] rule. The completed tree must have the predicates Q_1, Q_2, and S_3 defined, which come from the application of the [exp_p] rule. Fortunately, we already know that $Q_1 := \text{true}$ and $Q_3 := x = 0$, as those predicates come from the conclusion. The predicate Q_2 will be defined later.

Furthermore, the dictionary D needs to be defined. It should have the form $1 : Q_1,\ 2 : Q_2$. Since the value of Q_2 is not known yet, we temporarily define D as $D := 1 : \text{true},\ 2 : Q_2$.

$$\frac{\langle D \mid \{ \text{ true } \}\ S_1\ \{ \ Q_2 \ \}\rangle \ \text{(rule?)} \qquad \langle D \mid \{ \ Q_2 \ \}\ \texttt{skip}\ \{ \ x = 0 \ \}\rangle \ \text{(rule?)}}{\{ \text{ true } \}\ \texttt{1} : x := 0; \texttt{ goto 2}; \ x := 1 \ \&\ \texttt{2} : \texttt{skip}\ \{ \ x = 0 \ \}} \ [\text{exp}_\text{p}]$$

The second step is to find inference trees for $\langle D \mid \{ \text{ true } \}\ S_1\ \{ \ Q_2 \ \}\rangle$ and $\langle D \mid \{ \ Q_2 \ \}\ \texttt{skip}\ \{ \ x = 0 \ \}\rangle$. It is usually easier to find an inference tree for the last part, as the postcondition is known. Finding an inference tree for the first part may result in a value for Q_2 which does not fit the other parts of the inference tree.

One rule which can be applied to $\langle D \mid \{ \ Q_2 \ \}\ \texttt{skip}\ \{ \ x = 0 \ \}\rangle$ is the [skip_p] rule. As a result, we know that $Q_2 := x = 0$, thus $D := 1 : \text{true},\ 2 : x = 0$. The tree now looks like this:

$$\frac{\langle D \mid \{ \text{ true } \}\ S_1\ \{ \ x = 0 \ \}\rangle \ \text{(rule?)} \qquad \langle D \mid \{ \ x = 0 \ \}\ \texttt{skip}\ \{ \ x = 0 \ \}\rangle \ [\text{skip}_\text{p}]}{\{ \text{ true } \}\ \texttt{1} : x := 0; \texttt{ goto 2}; \ x := 1 \ \&\ \texttt{2} : \texttt{skip}\ \{ \ x = 0 \ \}} \ [\text{exp}_\text{p}]$$

The third step is to construct a tree with $\langle D \mid \{ \text{ true } \}\ S_1\ \{ \ x = 0 \ \}\rangle$ as its conclusion. Since S_1 is composed of three primitive statements, we will

separate this tree from our final result for the sake of readability. This tree has the following structure:

$$\frac{\langle D \mid \{\ \text{true}\ \}\ x := 0\ \{\ R_1\ \}\rangle \quad (\text{rule?}_1) \quad T_1}{\langle D \mid \{\ \text{true}\ \}\ x := 0;\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle}\ [\text{comp}_\text{p}]$$

Where T_1 is defined as:

$$\frac{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2}\ \{\ R_2\ \}\rangle \quad (\text{rule?}_2) \quad \langle D \mid \{\ R_2\ \}\ x := 1\ \{\ x = 0\ \}\rangle \quad (\text{rule?}_3)}{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle}\ [\text{comp}_\text{p}]$$

There are two options for rule?$_3$. One is to apply the [ass$_\text{p}$] rule, which results in the precondition $1 = 0$. Another option is to apply the [cons^2_p] rule. In this example, we use the latter option, which results in T_1 looking like this:

$$\frac{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2}\ \{\ \text{false}\ \}\rangle \quad (\text{rule?}_2) \quad \dfrac{\langle D \mid \{\ 1 = 0\ \}\ x := 1\ \{\ x = 0\ \}\rangle}{\langle D \mid \{\ \text{false}\ \}\ x := 1\ \{\ x = 0\ \}\rangle}\ \begin{matrix}[\text{ass}_\text{p}]\\ [\text{cons}^2_\text{p}]\end{matrix}}{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle}\ [\text{comp}_\text{p}]$$

This application of the [cons^2_p] rule is correct because false implies $1 = 0$, and $x = 0$ implies $x = 0$ (in this particular case, they are not only implications but also equality's).

As a result of the choice for rule?$_3$, the [goto$_\text{p}$] rule can be used for rule?$_2$. For the rule application to be valid, R_1 needs to be the predicate corresponding to label `2` in the dictionary D. By this point, we already know what that predicate is, namely $x = 0$. Thus T_1 looks like this:

$$\frac{\langle D \mid \{\ x = 0\ \}\ \texttt{goto 2}\ \{\ \text{false}\ \}\rangle \quad [\text{goto}_\text{p}] \quad \dfrac{\langle D \mid \{\ 1 = 0\ \}\ x := 1\ \{\ x = 0\ \}\rangle}{\langle D \mid \{\ \text{false}\ \}\ x := 1\ \{\ x = 0\ \}\rangle}\ \begin{matrix}[\text{ass}_\text{p}]\\ [\text{cons}^2_\text{p}]\end{matrix}}{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle}\ [\text{comp}_\text{p}]$$

Now that we know that $R_1 :=\ x = 0$, thus to complete the inference tree for $\langle D \mid \{\ \text{true}\ \}\ S_1\ \{\ x = 0\ \}\rangle$, we need to construct an inference tree with $\langle D \mid \{\ \text{true}\ \}\ x := 0\ \{\ x = 0\ \}\rangle$ as its conclusion. This can also be done with one application of the [cons$_\text{p}$] rule.

This results in a tree we call T:

$$\frac{\dfrac{\langle D \mid \{\ 0 = 0\ \}\ x := 0\ \{\ x = 0\ \}\rangle}{\langle D \mid \{\ \text{true}\ \}\ x := 0\ \{\ x = 0\ \}\rangle}\ \begin{matrix}[\text{ass}_\text{p}]\\ [\text{cons}_\text{p}]\end{matrix} \quad T_1}{\langle D \mid \{\ \text{true}\ \}\ x := 0;\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle}\ [\text{comp}_\text{p}]$$

Again, with $D := \texttt{1} : \text{true},\ \texttt{2} : x = 0$ and T_1 defined as:

$$\dfrac{\langle D \mid \{\ x = 0\ \}\ \texttt{goto 2}\ \{\ \text{false}\ \}\rangle\ [\text{goto}_\text{p}] \qquad \dfrac{\langle D \mid \{\ 1 = 0\ \}\ x := 1\ \{\ x = 0\ \}\rangle\ [\text{ass}_\text{p}]}{\langle D \mid \{\ \text{false}\ \}\ x := 1\ \{\ x = 0\ \}\rangle\ [\text{cons}^2_\text{p}]}}{\langle D \mid \{\ R_1\ \}\ \texttt{goto 2};\ x := 1\ \{\ x = 0\ \}\rangle\ [\text{comp}_\text{p}]}$$

Which results in the inference tree

$$\dfrac{T \qquad \langle D \mid \{\ x = 0\ \}\ \texttt{skip}\ \{\ x = 0\ \}\rangle\ [\text{skip}_\text{p}]}{\{\ \text{true}\ \}\ \texttt{1} : x := 0;\ \texttt{goto 2};\ x := 1\ \&\ \texttt{2} : \texttt{skip}\ \{\ x = 0\ \}}\ [\text{exp}_\text{p}]$$

A full version of this inference tree can be found in Appendix A.4.

6.3.2 Example 2

This program computes the faculty of x, with the assumption that $x \geq 0$, like the second example for SOS. We also define statements $S_1, S_{1,t}, S_2, S_{2,t}$ and S_3 to make the derivation sequence more readable.

$P :=$	$S_1 :=$
$\texttt{1} : \texttt{if}\ x \leq \texttt{1}\ \texttt{then}$	$\texttt{if}\ x \leq \texttt{1}\ \texttt{then}$
$\quad x := \texttt{1};$	$\quad S_{1,t}$
$\quad \texttt{goto 3}$	$\texttt{else}$
$\texttt{else}$	$\quad y := x\texttt{-1}$
$\quad y := x\texttt{-1}$	$S_{1,t} := x := \texttt{1}\ ;\ \texttt{goto 3}$
$\&\ \texttt{2} : \texttt{if}\ \neg(y\texttt{=1})\ \texttt{then}$	$S_2 := \texttt{if}\ \neg(y\texttt{=1})\ \texttt{then}$
$\quad x := x\texttt{*}y;$	$\quad S_{2,t}$
$\quad y := y\texttt{-1};$	$\texttt{else}$
$\quad \texttt{goto 2}$	$\quad \texttt{skip}$
$\texttt{else}$	$S_{2,t} := x := x\texttt{*}y\ ;\ y := y\texttt{-1}\ ;\ \texttt{goto 2}$
$\quad \texttt{skip}$	$S_3 := \texttt{skip}$
$\&\ \texttt{3} : \texttt{skip}$	

The following inference tree proofs that P calculates the faculty of x:

$$\dfrac{T_1 \quad T_2 \quad T_3}{\{\ x = n \wedge n \geq 0\ \}\ \texttt{1} : S_1\ \&\ \texttt{2} : S_2\ \&\ \texttt{3} : S_3\ \{\ x = n!\ \}}\ [\text{exp}_\text{p}]$$

With $D := \texttt{1} : x = n \wedge n \geq 0,\ \texttt{2} : x = n!/y!,\ \texttt{3} : x = n!$.

T_1 is defined as:

$$\dfrac{T_{1,t} \quad T_{1,f}}{\langle D \mid \{\ x = n \wedge n \geq 0\ \}\ S_1\ \{\ x = n!/y!\ \}\rangle}\ [\text{if}_\text{p}]$$

and $T_{1,t}$ is defined as:

$$\dfrac{T'_{1,t} \qquad \dfrac{\dfrac{}{\langle D \mid \{\ x = n!\ \}\ \texttt{goto 3}\ \{\ \text{false}\ \}\rangle}\,[\text{goto}_\text{p}]}{\langle D \mid \{\ x = n!\ \}\ \texttt{goto 3}\ \{\ x = n!/y!\ \}\rangle}\,[\text{cons}^2_\text{p}]}{\langle D \mid \{\ x \leq 1 \wedge x = n \wedge n \geq 0\ \}\ x := \texttt{1};\ \texttt{goto 3}\ \{\ x = n!/y!\ \}\rangle}\,[\text{comp}_\text{p}]$$

and $T'_{1,t}$ is defined as:

$$\dfrac{\dfrac{\dfrac{}{\langle D \mid \{\ 1 = n!\ \}\ x := \texttt{1}\ \{\ x = n!\ \}\rangle}\,[\text{ass}_\text{p}]}{\langle D \mid \{\ n = 0 \vee n = 1\ \}\ x := \texttt{1}\ \{\ x = n!\ \}\rangle}\,[\text{cons}^2_\text{p}]}{\langle D \mid \{\ x \leq 1 \wedge x = n \wedge n \geq 0\ \}\ x := \texttt{1}\ \{\ x = n!\ \}\rangle}\,[\text{cons}^2_\text{p}]$$

and $T_{1,f}$ is defined as:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle D \mid \{\ x = n!/(x-1)!\ \}\ y := x\texttt{-1}\ \{\ x = n!/y!\ \}\rangle}\,[\text{ass}_\text{p}]}{\langle D \mid \{\ x > 1 \wedge x = n \wedge x = x!/(x-1)!\ \}\ y := x\texttt{-1}\ \{\ x = n!/y!\ \}\rangle}\,[\text{cons}^2_\text{p}]}{\langle D \mid \{\ \neg(x \leq 1) \wedge x = n \wedge n \geq 0\ \}\ y := x\texttt{-1}\ \{\ x = n!/y!\ \}\rangle}\,[\text{cons}^2_\text{p}]$$

and T_2 is defined as:

$$\dfrac{T_{2,t} \qquad \dfrac{\dfrac{}{\langle D \mid \{\ y = 1 \wedge x = n!/y!\ \}\ \texttt{skip}\ \{\ y = 1 \wedge x = n!/y!\ \}\rangle}\,[\text{skip}_\text{p}]}{\langle D \mid \{\ \neg\neg(y = 1) \wedge x = n!/y!\ \}\ \texttt{skip}\ \{\ x = n!\ \}\rangle}\,[\text{cons}_\text{p}]}{\langle D \mid \{\ x = n!/y!\ \}\ S_2\ \{\ x = n!\ \}\rangle}\,[\text{if}_\text{p}]$$

and $T_{2,t}$ is defined as:

$$\dfrac{T'_{2,t} \qquad T''_{2,t}}{\langle D \mid \{\ \neg(y = 1) \wedge x = n!/y!\ \}\ S_{2,t}\ \{\ x = n!\ \}\rangle}\,[\text{comp}_\text{p}]$$

and $T'_{2,t}$ is defined as:

$$\dfrac{\dfrac{\dfrac{}{\langle D \mid \{\ x \cdot y = n!/(y-1)!\ \}\ x := x\texttt{*}y\ \{\ x = n!/(y-1)!\ \}\rangle}\,[\text{ass}_\text{p}]}{\langle D \mid \{\ x = n!/y!\ \}\ x := x\texttt{*}y\ \{\ x = n!/(y-1)!\ \}\rangle}\,[\text{cons}^2_\text{p}]}{\langle D \mid \{\ \neg(y = 1) \wedge x = n!/y!\ \}\ x := x\texttt{*}y\ \{\ x = n!/(y-1)!\ \}\rangle}\,[\text{cons}^2_\text{p}]$$

and $T''_{2,t}$ is defined as:

$$\dfrac{\dfrac{}{\langle D \mid \{\ x = n!/(y-1)!\ \}\ y := y\texttt{-1}\ \{\ x = n!/y!\ \}\rangle}\,[\text{ass}_\text{p}] \qquad T'''_{2,t}}{\langle D \mid \{\ x = n!/(y-1)!\ \}\ y := y\texttt{-1};\ \texttt{goto 2}\ \{\ x = n!\ \}\rangle}\,[\text{comp}_\text{p}]$$

and $T'''_{2,t}$ is defined as:

$$\dfrac{\dfrac{}{\langle D \mid \{\ x = n!/y!\ \}\ \texttt{goto 2}\ \{\ \text{false}\ \}\rangle}\,[\text{goto}_\text{p}]}{\langle D \mid \{\ x = n!/y!\ \}\ \texttt{goto 2}\ \{\ x = n!\ \}\rangle}\,[\text{cons}^2_\text{p}]$$

and T_3 is defined as:

$$\langle D \mid \{\ x = n!\ \}\ \texttt{skip}\ \{\ x = n!\ \}\rangle\ [\text{skip}_\text{p}]$$

A version of this inference tree where T_1, T_2, and T_3 are written in full can be found in Appendix A.5.

6.3.3 Example 3

This program is a simple example of a non-terminating program. With the way AS is defined for Goto, the postcondition is false in this case, although generally speaking AS cannot be used to prove that a program loops forever in this way.

$$P := \texttt{1: goto 1}$$

An inference tree with $\{\ \text{true}\ \}\ P\ \{\ \text{false}\ \}$ as its conclusion looks like this:

$$\dfrac{\langle \texttt{1:}\,\text{true} \mid \{\ \text{true}\ \}\ \texttt{goto 1}\ \{\ \text{false}\ \}\rangle\ [\text{goto}_\text{p}]}{\{\ \text{true}\ \}\ \texttt{1: goto 1}\ \{\ \text{false}\ \}}\ [\text{exp}_\text{p}]$$

6.3.4 Example 4

This program shows how non-normal programs (Definition 3.2.4) are handled in AS.

$$P := \texttt{1: skip; goto 2}$$

When creating an inference tree for P, one would get stuck at some point:

$$\dfrac{\dfrac{\langle \texttt{1:}Q_1 \mid \{\ Q_1\ \}\ \texttt{skip}\ \{\ Q_1\ \}\rangle\ [\text{skip}_\text{p}] \quad \langle \texttt{1:}Q_1 \mid \{\ Q_1\ \}\ \texttt{goto 2}\ \{\ Q_2\ \}\rangle\ [?]}{\langle \texttt{1:}Q_1 \mid \{\ Q_1\ \}\ \texttt{skip; goto 2}\ \{\ Q_2\ \}\rangle}\ [\text{comp}_\text{p}]}{\{\ Q_1\ \}\ \texttt{1: skip; goto 2}\ \{\ Q_2\ \}}\ [\text{exp}_\text{p}]$$

The only rule which can be applied at [?] is the consequence rule $[\text{cons}^2_\text{p}]$. However, this will not result in finishing this inference tree. The $[\text{goto}_\text{p}]$ rule can never be applied, since it requires an entry of the label the program should jump to in the dictionary. Since this goto statement jumps to label 2, but the dictionary has only an entry for label 1, it cannot be applied. Hence, one cannot create an inference tree for non-normal programs like P with these semantics.

6.4 Properties

For axiomatic semantics (AS), it is desirable to be able to prove that it is sound and complete. AS is sound, if only valid assertions can be proven, and AS is complete, if all valid assertions have a proof. De Bruin has proven soundness and completeness in [3] for the goto language he defined there, which is essentially the same language as Goto.

Before we properly define soundness and completeness, we define provability and validity of assertions first.

Definition 6.4.1 (Provable)**.** The assertions $\{Q_1\}P\{Q_2\}$ and $\langle D|\{Q_1\}S\{Q_2\}\rangle$ are called *provable*, if and only if there exists an inference tree with the respective assertion as its conclusion. This is written like this:

$$\vdash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$$
$$\vdash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$$

Definition 6.4.2 (Valid)**.** The assertions $\{Q_1\}P\{Q_2\}$ and $\langle D|\{Q_1\}S\{Q_2\}\rangle$ are called *valid*, if and only if:

1. (**Case** $\{Q_1\}\, P\, \{Q_2\}$ with $P = [n_i : S_i]_{i=1}^{k}$)
   for all states s and s', if Q_1 holds in state s and the derivation sequence $\langle P, S_1; \ldots; S_k, s\rangle \Rightarrow^* s'$ exists (using SOS rules as defined in Table 4.4), then Q_2 holds in state s'
2. (**Case** $\langle D \mid \{Q_a\}\, S\, \{Q_b\}\rangle$, with $D = [n_i : Q_i]_{i=1}^{k}$)
   for all states s, if Q_a holds in state s, then either
   - there exists a state s', such that $\mathcal{E}[\![S]\!]s = s'$ and Q_b is true in state s'
   - or there exists a state s' and a label-predicate pair $n_i : Q_i$ in D, such that $\mathcal{E}[\![S]\!]s = (s', n_i)$ and Q_i is true in state s'.

Validity is written like this:

$$\vDash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$$
$$\vDash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$$

Note. Due to the definition of validity, when proving soundness and completeness, we need to combine our AS for Goto with some form of operational semantics, i.e. SOS or NS. Since we consider the results of SOS and NS equal, we only consider SOS.

Definition 6.4.3 (Soundness)**.** Axiomatic semantics (AS) is *sound*, if all provable assertions are valid, i.e.

$$\vdash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\} \quad \text{implies} \quad \vDash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$$
$$\vdash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle \quad \text{implies} \quad \vDash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$$

Definition 6.4.4 (Completeness)**.** Axiomatic semantics (AS) is *complete*, if all valid assertions are provable, i.e.

$$\begin{aligned} &\vDash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\} && \text{implies } \vdash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\} \\ &\vDash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle && \text{implies } \vdash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle \end{aligned}$$

Conjecture 6.4.1. The axiomatic semantics for Goto is sound, i.e.
$\vdash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$ implies $\vDash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$ and
$\vdash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$ implies $\vDash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$

Conjecture 6.4.2. The axiomatic semantics for Goto is complete, i.e.
$\vDash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$ implies $\vdash_{\mathrm{p}} \{Q_1\}\, P\, \{Q_2\}$ and
$\vDash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$ implies $\vdash_{\mathrm{p}} \langle D \mid \{Q_1\}\, S\, \{Q_2\}\rangle$

Chapter 7

Related Work

This thesis does not include a chapter about denotational semantics for languages with goto. De Bruin has written a chapter about denotational semantics in [3], which is the paper we reference a lot in the structural operational semantics and axiomatic semantics chapter of this thesis.

Also, the language Goto we use here is (like the language While from Nielson and Nielson in [6]) made for use on a more conceptual level. For example, we do not specify how states should be implemented. Krebbers and Wiedijk included explicit memory management in the form of pointer allocation and deallocation in [7].

More practical research has been done on Typed Assembly Languages (TALs). TALs can be used to validate the safety of assembly-language programs. If such a program type checks, then the machine code of that program follows a predetermined safety policy. Ahmed, Appel, Richards, Swadi, Tan, and Wang have worked on how TALs can be proven sound in [1].

Chapter 8

Conclusions

In this paper, we have discussed how structural operational semantics (SOS), natural semantics (NS), and axiomatic semantics (AS) for languages with goto statements can be defined.

SOS makes use of a construct, which stores the entire program: $\langle P, S, s \rangle$. This is used to change the statement S, if a goto statement is executed. Furthermore, the composition rule for statements like $S_1; S_2$ is split into the five options for S_1. This is necessary for the goto rule to be able to change the entire S in $\langle P,\ S,\ s \rangle$.

As for NS, the derivation tree contains a series of transitions of the form $\langle P,\ pc,\ s \rangle \rightarrow s'$, accompanied by a *find* rule. This rule is used to get the transition of the form $\langle S,\ s \rangle \rightarrow \langle pc',\ c,\ s'' \rangle$, where S is the statement corresponding to the label pointed to by pc. This results in statements corresponding to certain labels being executed one by one, while c indicates if S has terminated at the normal exit point ($c = \circ$), or if there was a goto statement ($c = \bullet$).

Finally, AS makes use of a dictionary D, which contains tuples of labels and predicates. Every label and statement $n_i : S_i$ has one corresponding precondition Q_i, which must always be true, if a `goto` n_i statement is encountered (hence $\{\ Q_i\ \}$ `goto` n_i $\{$ false $\}$). Also, if any S_i terminates at the normal exit point, then Q_{i+1} must hold.

Bibliography

[1] Amal Ahmed and Andrew W. Appel and Christopher D. Richards and Kedar N. Swadi and Gang Tan and Daniel C. Wang. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.*, 32(3):7:1–7:67, 2010.

[2] Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958.

[3] Arie de Bruin. Goto Statements: Semantics and Deduction Systems. *Acta Informatica*, 15:385–424, 1981.

[4] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[5] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[6] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications – a formal introduction*. Wiley professional computing. Wiley, 1992.

[7] Robbert Krebbers and Freek Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7794 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013.

[8] Frank Rubin. "GOTO Considered Harmful" Considered Harmful. *Commun. ACM*, 30(3):195–196, 1987.

[9] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, 373(3):273–302, 2007.

Appendix A

Appendix

A.1 NS derivation trees

A.1.1 Example 1

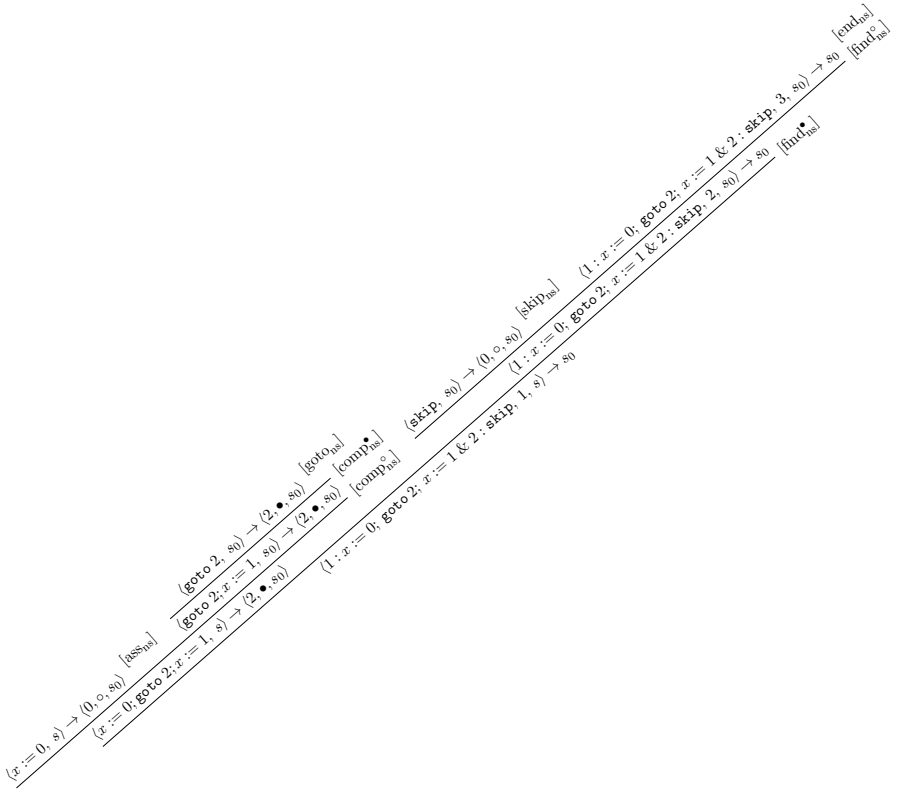


Figure A.1: The full derivation tree for Example 5.2.1

A.1.2 Example 2 (s x = 0)

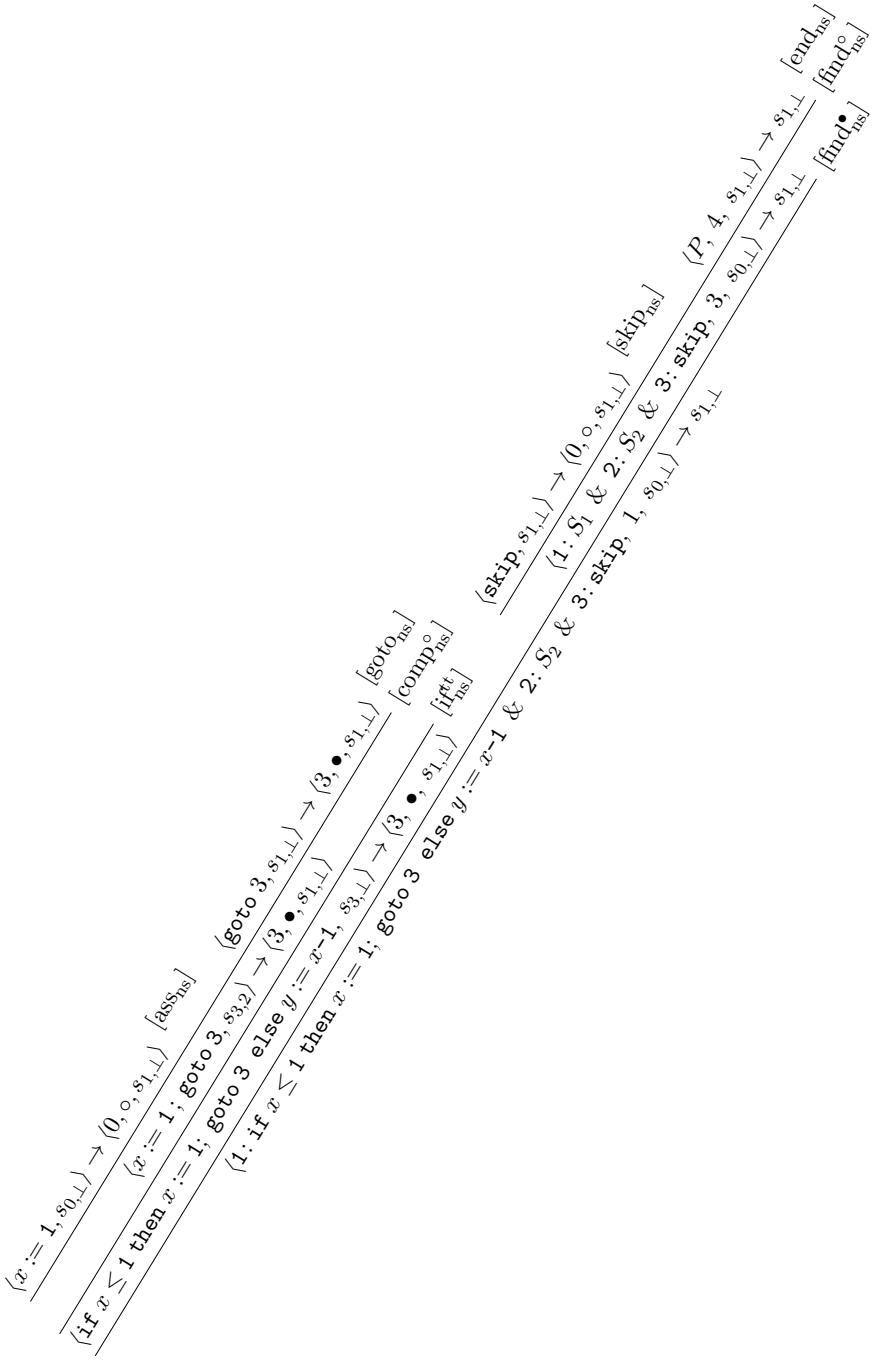


Figure A.2: A full derivation tree for Example 5.2.2 with abbreviations

A.1.3 Example 2 (s x = 3)

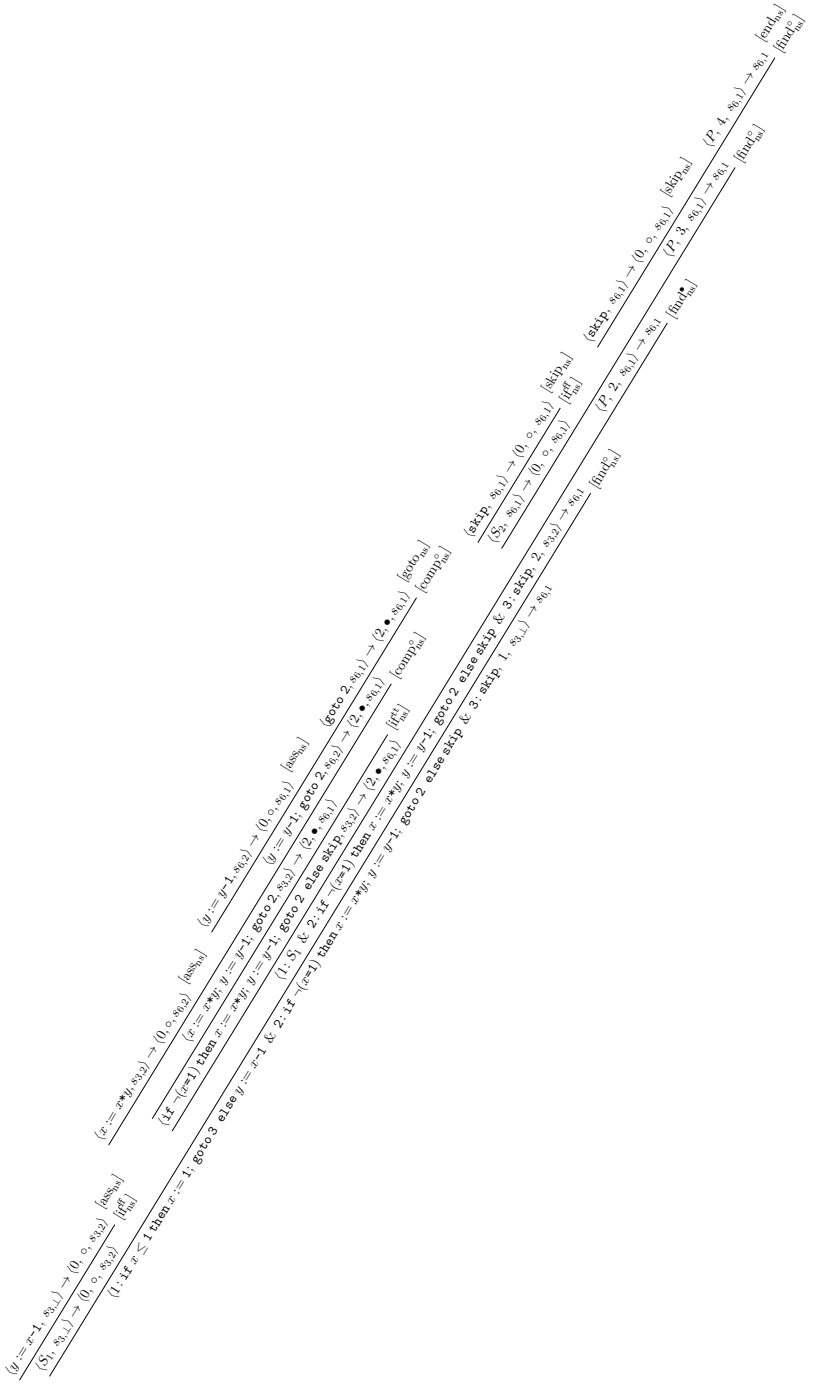


Figure A.3: A full derivation tree for Example 5.2.2 with abbreviations

A.2 AS inference trees

A.2.1 Example 1

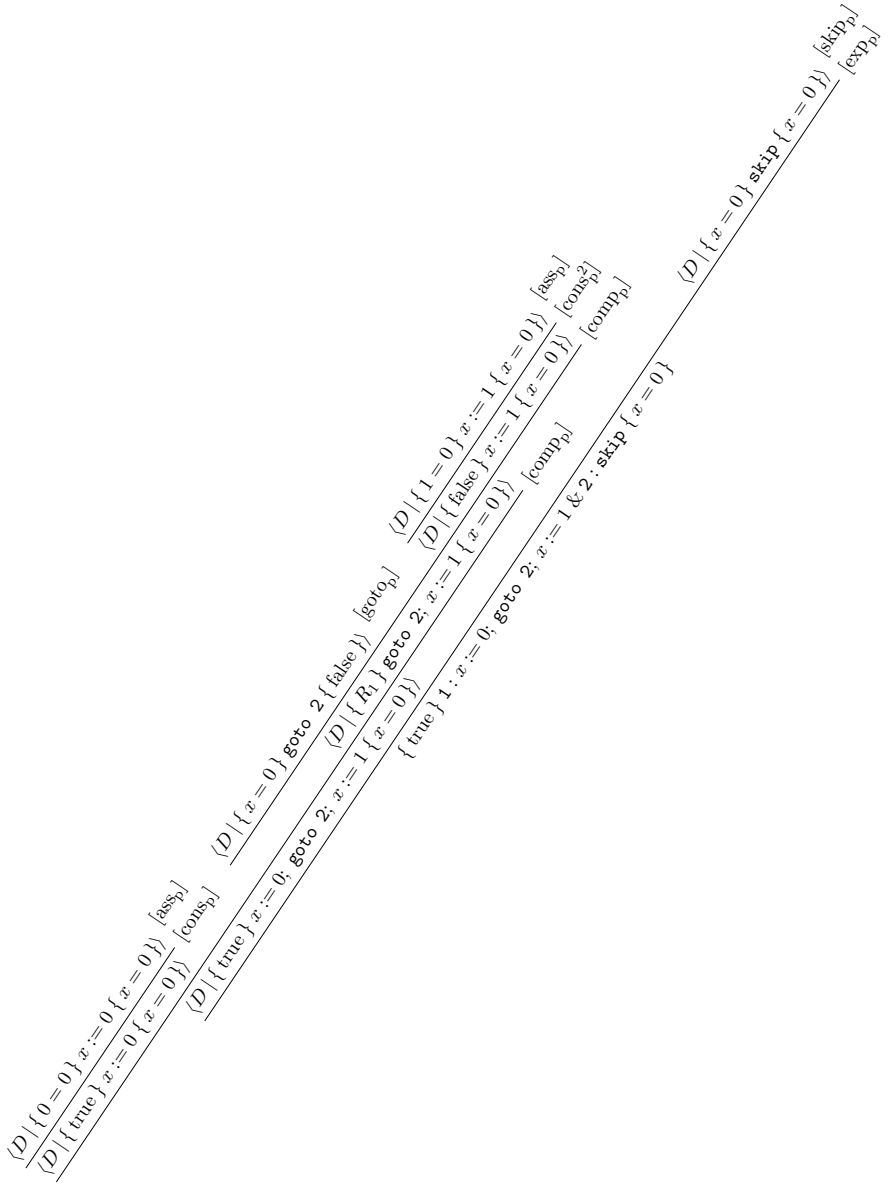


Figure A.4: The full inference tree for Example 6.3.1 with $D = 1{:}\text{true},\ 2{:}x = 0$

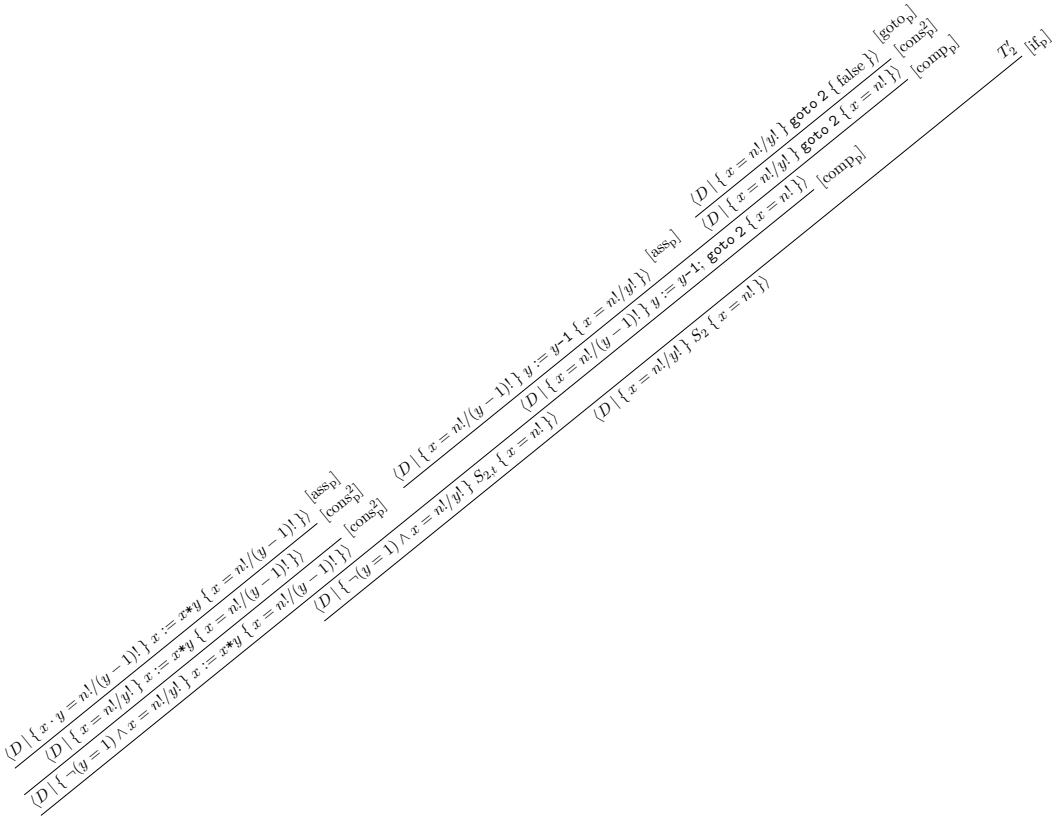
A.2.2 Example 2

$$\frac{T_1 \quad T_2 \quad T_3}{\{\, x = n \wedge n \geq 0 \,\}\ \mathtt{1} : S_1 \ \&\ \mathtt{2} : S_2 \ \&\ \mathtt{3} : S_3\ \{\, x = n! \,\}}\ [\mathrm{exp_p}]$$

Where T_1 is defined as:

$$\dfrac{\dfrac{\dfrac{\langle D \mid \{\, 1 = n! \,\}\ x := \mathtt{1}\ \{\, x = n! \,\}\rangle}{\langle D \mid \{\, n = 0 \vee n = 1 \,\}\ x := \mathtt{1}\ \{\, x = n! \,\}\rangle}\ [\mathrm{cons}^2_{\mathrm{p}}]}{\langle D \mid \{\, x \leq 1 \wedge x = n \wedge n \geq 0 \,\}\ x := \mathtt{1}\ \{\, x = n! \,\}\rangle}\ [\mathrm{cons}^2_{\mathrm{p}}] \quad \dfrac{\langle D \mid \{\, x = n! \,\}\ \mathtt{goto\ 3}\ \{\, \mathrm{false} \,\}\rangle}{\langle D \mid \{\, x = n! \,\}\ \mathtt{goto\ 3}\ \{\, x = n!/y! \,\}\rangle}\ [\mathrm{cons}^2_{\mathrm{p}}]}{\langle D \mid \{\, x \leq 1 \wedge x = n \wedge n \geq 0 \,\}\ x := \mathtt{1};\ \mathtt{goto\ 3}\ \{\, x = n!/y! \,\}\rangle}\ [\mathrm{comp_p}] \quad \dfrac{\dfrac{\langle D \mid \{\, x = n!/(x-1)! \,\}\ y := x\mathtt{-1}\ \{\, x = n!/y! \,\}\rangle}{\langle D \mid \{\, x > 1 \wedge x = n \wedge x = x!/(x-1)! \,\}\ y := x\mathtt{-1}\ \{\, x = n!/y! \,\}\rangle}\ [\mathrm{cons}^2_{\mathrm{p}}]}{\langle D \mid \{\, \neg(x \leq 1) \wedge x = n \wedge n \geq 0 \,\}\ y := x\mathtt{-1}\ \{\, x = n!/y! \,\}\rangle}\ [\mathrm{cons}^2_{\mathrm{p}}]}{\langle D \mid \{\, x = n \wedge n \geq 0 \,\}\ S_1\ \{\, x = n!/y! \,\}\rangle}\ [\mathrm{if_p}]$$

T_2 is defined as:



T_2' is defined as:

$$\frac{\langle D \mid \{\, y = 1 \wedge x = n!/y! \,\}\ \mathtt{skip}\ \{\, y = 1 \wedge x = n!/y! \,\}\rangle}{\langle D \mid \{\, \neg\neg(y = 1) \wedge x = n!/y! \,\}\ \mathtt{skip}\ \{\, x = n! \,\}\rangle}\ \begin{matrix}[\mathrm{skip_p}] \\ [\mathrm{cons_p}]\end{matrix}$$

and T_3 is defined as:

$$\langle D \mid \{\, x = n! \,\}\ \mathtt{skip}\ \{\, x = n! \,\}\rangle\ [\mathrm{skip_p}]$$

Figure A.5: The inference tree for Example 6.3.2 with $D = \mathtt{1}{:}x = n \wedge n \geq 0,\ \mathtt{2}{:}x = n!/y!,\ \mathtt{3}{:}x = n!$