Side Effects in Steering Fragments

MSc Thesis (*Afstudeerscriptie*)

written by

L.L. Wortel
(born 12 October 1984 in Heemskerk)

under the supervision of **Alban Ponse** and **Paul Dekker**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
September 5th, 2011	Dr Alban Ponse
	Dr Paul Dekker
	Prof Dr Jan van Eijck
	Dr Sara Uckelman
	Prof Dr Benedikt Löwe

Institute for Logic, Language and Computation

I'm dedicating this thesis to my parents, without whom I would never have gotten to this point. I have tested their patience by taking my time to graduate, but they kept supporting every silly little thing I have ever done. Thanks guys, you're the best.

Abstract

In this thesis I will give a formal definition of side effects. I will do so by modifying a system for modelling program instructions and program states, Quantified Dynamic Logic, to a system called $\mathrm{DLA_f}$ (for Dynamic Logic with Assignments as Formulas), which in contrast to QDL allows assignments in formulas and makes use of short-circuit evaluation. I will show the underlying logic in those formulas to be a variant of short-circuit logic called repetition-proof short-circuit logic.

Using $\mathrm{DLA_f}$ I will define the actual and the expected evaluation of a single instruction. The side effects are then defined to be the difference between the two. I will give rules for composing those side effects in single instructions, thus scaling up our definition of side effects to a definition of side effects in deterministic $\mathrm{DLA_f}$-programs. Using this definition I will give a classification of side effects, introducing as most important class that of marginal side effects. Finally, I will show how to use our system for calculating the side effects in a real system such as Program Algebra (PGA).

Acknowledgements

I would first and foremost like to thank my supervisor Alban Ponse for the big amounts of time and energy he put into guiding me through this project. His advice has been invaluable to me and his enthusiasm has been a huge motivation for me throughout.

A thank you also goes out to Jan van Eijck for pointing me in the right direction halfway through the project.

Finally I would like to thank my entire thesis committee, consisting of Alban Ponse, Paul Dekker, Jan van Eijck, Sara Uckelman and Benedikt Löwe, for taking the time to read and grade my thesis.

— Lars Wortel, August 2011

Contents

1 Introduction

1.1 What are side effects?

In programming practice, *side effects* are a well-known phenomenon, even though nobody seems to have an exact definition of what they are. To get a basic idea, here are some examples from natural language and programming that should explain the intuition behind side effects.

Suppose you and your wife have come to an agreement regarding grocery shopping. Upon leaving for work, she told you that "if I don't call, you do not have to do the shopping". Later that day, she calls you to tell you something completely different, for instance that she is pregnant. This call now has as side effect that you no longer know whether you have to do grocery shopping or not, even though the meaning of the call itself was something completely different.

Another example is taken from [9]. Suppose someone tells you that "Phoebe is waiting in front of your door, and you don't know it!" This is a perfectly fine thing to say, but you cannot say it twice because then it will no longer be true that you don't know that Phoebe is waiting (after all, you were just told). Here, the side effect is that your knowledge gets updated by the sentence, which makes the latter part of that sentence, which is a statement about your knowledge, false.

As said, in programming practice, side effects are a well-known phenomenon. Logically, they are interesting because the possible presence of side effects in a program instruction sequence invalidates principles of propositional logic such as commutativity ($\phi \wedge \psi \leftrightarrow \psi \wedge \phi$) and idempotency ($\phi \wedge \phi \leftrightarrow \phi$). The textbook example is the following program:

```
x:=1
if (x:=x+1 and x=2) then y
```

Here the operator := stands for assignment and = for an equality test. Assuming an assignment instruction always succeeds (that is, yields the reply `true`), in the above example the test $\phi \wedge \psi$, where ϕ is the instruction `x:=x+1` and ψ the instruction `x=2`, will succeed and therefore, y will be executed. However, should the order of those instructions be reversed ($\psi \wedge \phi$), this no longer will be the case. The reason is that the instruction ϕ has a side effect: apart from returning

`true`, it also increments the variable x with 1, thus making it 2. If ϕ is executed before ψ, the test in ψ (`x=2`) will yield `true`. Otherwise, it will yield `false`.

It is easy to see that should $\phi \wedge \psi$ be executed twice, the end result will also be `false`. Therefore, for $\chi = \phi \wedge \psi$, we have that $\chi \wedge \chi \not\leftrightarrow \chi$.

1.2 What are steering fragments?

Now that I have given a rough idea of what side effects are, the reader is probably wondering about the second part of my thesis title: that of steering fragments. A *steering fragment* or *test* is a program fragment which is concerned with the control flow of the execution of that program. To be exact, a steering fragment will use the evaluation result of a formula (which is a Boolean) and depending on the outcome, will steer further execution of the program. Thus, a steering fragment consists of two parts: a formula and a control part which decides what to do with the evaluation result of that formula. Throughout this thesis, I will be using the terms steering fragment and test interchangeably.

The formula in a steering fragment can either be a primitive or a compound formula. The components of a compound formula are usually connected via logical connectives such as $\wedge$ and $\vee$, or involve negation. If the formula of a steering fragment is compound, we say that the steering fragment is a *complex steering fragment*.

We have already seen a classical example of a (complex) steering fragment in the previous section: the *if* ... *then* instruction. In the example above, the formula is a compound formula with $x := x + 1$ and $x = 2$ as its components, connected via the logical connective $\wedge$. The control part of this steering fragment consists of *if* and *then* and the prescription to execute y if evaluation of `x:=x+1 and x=2` yields true.

1.3 Related work

The main contribution of this thesis is to construct a formal model of side effects in dynamic logic. Because of that, I only had limited time and space to properly research related work done in this area. Despite that, I will briefly describe some references I have come across throughout this project.

Currently, a formal definition of side effects appears to be missing in literature. That is not to say that side effects have been completely ignored. Attempts have been made to create a logic which admits the possibility of side effects by Bergstra and Ponse [5]. Furthermore, an initial, informal classification of side effects has been presented by Bergstra in [1]. I will return to those references later in this thesis.

Black and Windley have made an attempt to reason in a setting with side effects in [7, 8]. In their goal to verify a secure application written in C using Hoare axiomatic semantics to express the correctness of program statements, they encountered the problem of side effects occurring in the evaluation of some C-expressions. They solved the problem by creating extra inference rules which essentially separate the evaluation of the side effect from the evaluation of the main expression.

Also working with C is Norrish in [17]. He presents a formal semantics for C and he, too, runs into side effects in the process. Norrish claims that a semantics gives a program meaning by describing the way in which it changes a program state. Such a program state would both include the computer's memory as well as what is commonly known as the environment (types of variables, mapping of variable names to addresses in memory etc.). Norrish claims that in C, changes to the former come about through the actions of side effects, which are created by evaluating certain expression forms such as assignments. Norrish' formal semantics for C is able to handle these side effects.

Böhm presents a different style of axiomatic definitions for programming languages [6]. Whereas other authors such as Black and Windley above use Hoare axiomatic semantics which bases the logic on the notion of pre- or postcondition, Böhm uses the value of a programming language expression as the underlying primitive. He relies on the fact that the underlying programming language is an expression language such as Algol 68 [21]. Expressions are allowed to have arbitrary side effects and the notions of statement and expression coincide. Böhm claims that his formalism is just as intuitive as Hoare-style logic and that the notion of 'easy axiomatizability' — which is a major measurement of the quality of a programming language — is a matter of a choice of formalism, which in turn is arbitrary.

In this thesis I will develop a variant of Dynamic Logic to model side effects. Dynamic Logic is used for a wide range of applications, ranging from modelling key constructs of imperative programming to developing dynamic semantic theories for natural language. An early overview of dynamic logic is given by Harel in [15]. More recently, Van Eijck and Stokhof have given an extensive overview of various systems of dynamic logic in [11].

1.4 Overview of this thesis

Intuitively, a side effect of a propositional statement is a change in state of a program or model other than the effect (or change in state) it was initially executed for. In this thesis I will present a system that makes this intuition explicit.

First, in Chapter 2 I will present the preliminaries on which my system, that can model program instructions and their effect on program states, is based. This system, which I present in Chapter 3, will be a modified version of Quantified Dynamic Logic, overviews of which can be found in [15, 11].

After introducing some terminology and exploring the logic behind this system in Chapters 4 and 5, I can formally define side effects, which I will do in Chapter 6. In Chapter 7 I will proceed to giving a classification of side effects, introducing marginal side effects as the most important class.

In Chapter 8 I will present a case study to see this definition of side effects in action. For this I will use an — again slightly modified — version of Program Algebra [3]. I will end this thesis with some conclusions and some pointers for future work.

2 Preliminaries

2.1 Introduction

In order to say something useful about side effects, we need a formal definition. Such a definition can be found using dynamic logics. The basic idea here is that the update of a program instruction is the change in program state it causes. This allows us to introduce an expected and an actual evaluation of a program instruction. The expected evaluation of a program instruction is the change you would expect a program instruction to make to the program state upon evaluation. This may differ, however, from the actual evaluation, namely when a side effect occurs when actually evaluating the program instruction. The side effect of a program instruction then is defined as the difference in expected and actual evaluation of a program instruction.

To flesh this out in a formal definition, we first need a system that is able to model program states and program instructions. *Quantified Dynamic Logic* (QDL) is such a system. QDL was developed by Harel [14] and Goldblatt [13]. It can be seen as a first order version of *Propositional Dynamic Logic* (PDL), which was developed by Pratt in [19, 20]. Much of the overview of both PDL and QDL I will give below is taken from the overview of dynamic logic by Van Eijck and Stokhof [11].

Dynamic logic can be viewed as dealing with the logic of action and the result of action [11]. Although various kinds of actions can be modelled with it, one is of particular interest for us: the actions performed on computers, i.e. computations. In essence, these are actions that change the memory state of a machine, or on a somewhat higher level the program state of a computer program.

Regardless of what kinds of actions are modelled, the core of dynamic logic can in many cases be characterized in a similar way via the logic of 'labelled transistion systems'. A labelled transition system or LTS over a signature $\langle P, A\rangle$, with P a set of propositions and A a set of actions, is a triple $\langle S, V, R\rangle$ where S is a set of states, $V : S \to \mathcal{P}(P)$ is a valuation function and $R = \{\stackrel{a}{\to} \subseteq S \times S \mid a \in A\}$ is a set of labelled transitions (one binary relation on S for each label a).

There are various versions of dynamic logic. Before I will introduce two of

these, I will first describe the setting I will be using in my examples. This setting consists of a toy programming language that is expressive enough to model the working examples I need to discuss side effects.

2.2 Toy language

My toy language should be able to handle assignments and steering fragments. The steering fragment can possibly be complex, so our toy language should be able to handle compound formulas: multiple formulas (such as equality tests) connected via logical connectives. In particular, I will be using short-circuit left and ($\overset{\circ}{\wedge}$) and short-circuit left or ($\overset{\circ}{\vee}$) as connectives. Finally, assignments should be allowed in tests as well: they are, in line with what one would expect, defined to always return `true`.

As toy language I will first present the WHILE language defined by Van Eijck in [11]. We will see soon enough that we will actually need more functionality than it offers, but it will serve us well in the introduction of PDL, QDL and the illustration of the problems we will run into.

The WHILE language works on natural numbers and defines arithmetic expressions, Boolean expressions and programming commands. Arithmetic expressions a with n ranging over numerals and v over variables from a set $\mathcal{V}$ are defined as follows:

$$a ::= n \mid v \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 \dot{-} a_2$$

Boolean expressions are defined as:

$$B ::= \top \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg B \mid B_1 \vee B_2$$

Finally, we define the following programming commands:

$$C ::= \text{SKIP} \mid \text{ABORT} \mid v := a \mid C_1; C_2 \mid \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2$$

For the sake of simplicity, we will postpone the introduction of the WHILE command until after we have presented our modified system in Chapter 3.

The semantics of the arithmetic expressions are fairly self-explanatory. We assume that every numeral n in N has an interpretation $I(n) \in \mathbb{N}$ and let g be a mapping from $\mathcal{V}$ to $\mathbb{N}$. We then have the following interpretations of the arithmetic expressions, relative to initial valuation or initial program state g:

$$\begin{aligned}
[\![n]\!]_g &:= I(n) \\
[\![v]\!]_g &:= g(v) \\
[\![a_1 + a_2]\!]_g &:= [\![a_1]\!]_g + [\![a_2]\!]_g \\
[\![a_1 * a_2]\!]_g &:= [\![a_1]\!]_g * [\![a_2]\!]_g \\
[\![a_1 \dot{-} a_2]\!]_g &:= [\![a_1]\!]_g \dot{-} [\![a_2]\!]_g
\end{aligned}$$

The semantics of the Boolean expressions are standard as well, writing T for

true and F for false:

$$\begin{aligned}
[\![\top]\!]_g &:= T \\
[\![a_1 = a_2]\!]_g &:= \begin{cases} T \text{ if } [\![a_1]\!]_g = [\![a_2]\!]_g \\ F \text{ otherwise} \end{cases} \\
[\![a_1 \leq a_2]\!]_g &:= \begin{cases} T \text{ if } [\![a_1]\!]_g \leq [\![a_2]\!]_g \\ F \text{ otherwise} \end{cases} \\
[\![\neg B]\!]_g &:= \begin{cases} T \text{ if } [\![B]\!]_g = F \\ F \text{ otherwise} \end{cases} \\
[\![B_1 \vee B_2]\!]_g &:= \begin{cases} T \text{ if } [\![B_1]\!]_g = T \text{ or } [\![B_2]\!]_g = T \\ F \text{ otherwise} \end{cases}
\end{aligned}$$

The semantics of the commands of the toy language can be given in various styles. Here I take a look at a variant called structural operational semantics [11]. It is specified using a transition system from pairs of a state and a command, to either a state or again a state and a (new) command.

First I will give the transitions for the assignment command. It looks like this, where we write $g[v \mapsto t]$ for the valuation which is like valuation g except for the variable v, which has been mapped to t:

$$(g, v := t) \Longrightarrow g[v \mapsto [\![t]\!]_g]$$

Here we have the pair of state g and the assignment command $v := a$ at the start of the transition. After the transition, we only have a new state left, since the execution of this command has finished in a single step.

The SKIP command does nothing: it does not change the state and it finishes in a single step.

$$(g, \text{SKIP}) \Longrightarrow g$$

In structural operational semantics, there are two rules for sequential composition, one for when program C_1 finishes in a single step and one for which it does not.

$$\frac{(g, C_1) \Longrightarrow g'}{(g, C_1; C_2) \Longrightarrow (g', C_2)}$$

$$\frac{(g, C_1) \Longrightarrow (g', C_1')}{(g, C_1; C_2) \Longrightarrow (g', C_1'; C_2)}$$

Finally, we have the rules for conditional action. There are two (similar) rules, depending on the outcome of the test:

$$\frac{}{(g,\ \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) \Longrightarrow (g, C_1)} [\![B]\!]_g = T$$

$$\frac{}{(g,\ \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) \Longrightarrow (g, C_2)} [\![B]\!]_g = F$$

2.3 Propositional Dynamic Logic

Now that I have introduced the toy language, it is time to take a look at the first version of dynamic logic we are interested in: Propositional Dynamic Logic (PDL in short). The language of PDL consists of formulas ϕ (based on basic propositions $p \in P$) and programs α (based on basic actions $a \in A$):

$$\begin{aligned}\phi &::= \top \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \langle\alpha\rangle\phi \\ \alpha &::= a \mid ?\phi \mid \alpha_1;\alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^*\end{aligned}$$

As the name suggests, PDL is based on propositional logic. This means that the usual properties such as associativity and duality are valid and will be used throughout. Furthermore, we can use the following abbreviations:

$$\begin{aligned}\bot &= \neg\top \\ \phi_1 \wedge \phi_2 &= \neg(\neg\phi_1 \vee \neg\phi_2) \\ \phi_1 \to \phi_2 &= \neg\phi_1 \vee \phi_2 \\ \phi_1 \leftrightarrow \phi_2 &= (\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1) \\ [\alpha]\phi &= \neg\langle\alpha\rangle\neg\phi\end{aligned}$$

The relational composition $R_1 \circ R_2$ of binary relations R_1, R_2 on state set S is given by:

$$R_1 \circ R_2 = \{(t_1, t_2) \in S \times S \mid \exists t_3((t_1, t_3) \in R_1 \wedge (t_3, t_2 \in R_2))\}$$

The n-fold composition R^n of a binary relation R on S with itself is recursively defined as follows, with I the identity relation on S:

$$\begin{aligned}R^0 &= I \\ R^n &= R \circ R^{n-1}\end{aligned}$$

Finally, the reflexive transitive closure of R is given by:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n$$

To define the semantics of PDL over basic propositions P and basic actions A, we need the labelled transistion system $T = \langle S_T, V_T, R_T\rangle$ for signature $\langle P, A\rangle$. The formulas of PDL are interpreted as subsets of S_T, the actions as binary

relations on S_T. This leads to the following interpretations:

$$\begin{aligned}
[\![\top]\!]^T &= S_T \\
[\![p]\!]^T &= \{s \in S_T \mid p \in V_T(s)\} \\
[\![\neg\phi]\!]^T &= S_T - [\![\phi]\!]^T \\
[\![\phi_1 \vee \phi_2]\!]^T &= [\![\phi_1]\!]^T \cup [\![\phi_2]\!]^T \\
[\![\langle\alpha\rangle\phi]\!]^T &= \{s \in S_T \mid \exists t (s,t) \in [\![\alpha]\!]^T \text{ and } t \in [\![\phi]\!]^T\}
\end{aligned}$$

$$\begin{aligned}
[\![a]\!]^T &= \xrightarrow{a}_T \\
[\![?\phi]\!]^T &= \{(s,s) \in S_T \times S_T \mid s \in [\![\phi]\!]^T\} \\
[\![\alpha_1;\alpha_2]\!]^T &= [\![\alpha_1]\!]^T \circ [\![\alpha_2]\!]^T \\
[\![\alpha_1 \cup \alpha_2]\!]^T &= [\![\alpha_1]\!]^T \cup [\![\alpha_2]\!]^T \\
[\![\alpha^*]\!]^T &= ([\![\alpha]\!]^T)^*
\end{aligned}$$

The programming constructs in our toy language are expressed in PDL as follows:

$$\begin{aligned}
\text{SKIP} &:= ?\top \\
\text{ABORT} &:= ?\bot \\
\text{IF } \phi \text{ THEN } \alpha_1 \text{ ELSE } \alpha_2 &:= (?\phi;\alpha_1) \cup (?\neg\phi;\alpha_2)
\end{aligned}$$

Although PDL is a powerful logic, it is not enough yet to properly model the toy language we need. The reason for that is the need for assignments. Since assignments change relational structures, the appropriate assertion language is first order predicate logic, and not propositional logic [11]. So instead of PDL, which as the name suggests uses propositional logic, we need a version of dynamic logic that uses first order predicate logic. This is where Quantified Dynamic Logic (QDL in short) comes in.

2.4 Quantified Dynamic Logic

The language of QDL consists of terms t, formulas ϕ and programs π. For functions f and relational symbols R we have:

$$\begin{aligned}
t &::= v \mid ft_1 \dots t_n \\
\phi &::= \top \mid Rt_1 \dots t_n \mid t_1 = t_2 \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists v\phi \mid \langle\pi\rangle\phi \\
\pi &::= v := ? \mid v := t \mid ?\phi \mid \pi_1;\pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^*
\end{aligned}$$

In the case of natural numbers, examples of f are $+, *$ etc. and examples of R are $\leq$ and $\geq$. The same abbreviations as in PDL are used, most notably $\bot = \neg\top$ and $[\pi]\phi = \neg\langle\pi\rangle\neg\phi$.

The random assignment $(v := ?)$ does not increase the expressive power of QDL [11]. It can, however, be nicely used to express the universal and existential quantifier:

$$\begin{aligned}
\exists v\phi &\leftrightarrow \langle v := ?\rangle\phi \\
\forall v\phi &\leftrightarrow [v := ?]\phi
\end{aligned}$$

The pair (f, R) is called a first order signature. A model for such a signature is a structure of the form

$$M = (E^M, f^M, R^M)$$

where E is a non-empty set, the f^M are interpretations in E for the members of f and the R^M similarly are the interpretations in E for the members of R. Now let $\mathcal{V}$ be the set of variables of the language. Interpretation of terms in M is defined relative to an initial valuation $g : \mathcal{V} \to E^M$:

$$[\![v]\!]^M_g = g(v) \tag{QDL1}$$
$$[\![ft_1 \dots t_n]\!]^M_g = f^M([\![t_1]\!]^M_g, \dots, [\![t_n]\!]^M_g) \tag{QDL2}$$

Truth in M for formulas is defined by simultaneous recursion, where $g \sim_v h$ then means that h differs at most from g on the assignment it gives to variable v:

$$M \models_g \top \text{ always} \tag{QDL3}$$
$$M \models_g Rt_1 \dots t_n \text{ iff } ([\![t_1]\!]^M_g, \dots, [\![t_n]\!]^M_g) \in R^M \tag{QDL4}$$
$$M \models_g t_1 = t_2 \text{ iff } [\![t_1]\!]^M_g = [\![t_2]\!]^M_g \tag{QDL5}$$
$$M \models_g \neg\phi \text{ iff } M \not\models_g \phi \tag{QDL6}$$
$$M \models_g \phi_1 \vee \phi_2 \text{ iff } M \models_g \phi_1 \text{ or } M \models_g \phi_2 \tag{QDL7}$$
$$M \models_g \exists v\phi \text{ iff for some } h \text{ with } g \sim_v h, M \models_h \phi \tag{QDL8}$$
$$M \models_g \langle\pi\rangle\phi \text{ iff for some } h \text{ with } {}_g[\![\pi]\!]^M_h, M \models_h \phi \tag{QDL9}$$

The same goes for the relational meaning in M for programs:

$${}_g[\![v := t]\!]^M_h \text{ iff } h = g[v \mapsto [\![t]\!]^M_g] \tag{QDL10}$$
$${}_g[\![?\phi]\!]^M_h \text{ iff } g = h \text{ and } M \models_g \phi \tag{QDL11}$$
$${}_g[\![\pi_1; \pi_2]\!]^M_h \text{ iff } \exists f \text{ with } {}_g[\![\pi_1]\!]^M_f \text{ and } {}_f[\![\pi_2]\!]^M_h \tag{QDL12}$$
$${}_g[\![\pi_1 \cup \pi_2]\!]^M_h \text{ iff } {}_g[\![\pi_1]\!]^M_h \text{ or } {}_g[\![\pi_2]\!]^M_h \tag{QDL13}$$
$${}_g[\![\pi^*]\!]^M_h \text{ iff } g = h \text{ or } {}_g[\![\pi; \pi^*]\!]^M_h \tag{QDL14}$$

The above definition makes concatenation (;) an associative operator:

$$(\pi_1; \pi_2); \pi_3 = \pi_1; (\pi_2; \pi_3)$$

As a convention, we omit the brackets wherever possible.

Although QDL goes a long way to modelling our toy language and program states, we are not quite there yet. The modifications we have to make come to light when we examine the expressive power of QDL. QDL currently has more expressive power than it has semantics defined for. This problem surfaces when the modality operator is nested within a test, like this:

$$?(\langle v := t\rangle\top)$$

This is the program $?\phi$, with $\phi = \langle\pi\rangle\psi$, $\pi = v := t$ and $\psi = \top$. As the semantics of QDL are currently defined, the program π will make a change to an initial

valuation g if it is interpreted in it, returning valuation h where the assignment g had for variable v will be expressed by t. This is expressed by QDL10. However, the current semantics only assign relational meaning to a test instruction $?\phi$ as long as $g = h$, as expressed by QDL11.

Another similar example is the following:

$$?(\langle v := v + 1; v := v \dot{-} 1\rangle \top)$$

Although this situation should be similar as above, it is not: because the program state gets changed twice, QDL now *is* able to assign semantics to this program since the program state gets returned to the original state by the second program instruction (and we therefore have $g = h$).

So, not only can we devise even a very simple correct QDL-program for which there are no semantics defined, we can also give a very similar example for which QDL does define semantics. Not only does that somewhat erratic behavior seem undesirable, but the nature of the examples here present us with a problem when we are considering side effects. Exactly for the situations in which side effects occur, namely when an instruction in a test causes a change in the program state, there are no semantics defined in QDL. Therefore, I am going to have to modify QDL so that it does define semantics in those situations.

3

Modifying QDL to $\mathbf{DLA_f}$

3.1 Introducing $\mathbf{DLA_f}$

In this chapter I will present *Dynamic Logic with Assignments as Formulas*, or $\mathrm{DLA_f}$ in short, the resulting dynamic logic after making two major modifications to QDL. The modifications I will make are such that $\mathrm{DLA_f}$ can model the specific kinds of constructions that we are interested in. This means that, like the name suggests, we have to introduce semantics for assignments in formulas. Furthermore, we will drop or modify some other QDL-instructions that we do not need. Because of that $\mathrm{DLA_f}$ evades the problem of QDL mentioned in Section 2.4 of the previous chapter and one other problem I will get back to in Section 3.3. Before I introduce $\mathrm{DLA_f}$, however, I will show the modifications that need to be done to Van Eijck's WHILE language so that it can model the instructions we need.

In the WHILE language, Boolean expressions are assumed to cause no state change upon evaluation. However, for our purpose this is inadequate. We want to allow assignments in tests as well and they cause a state change. This warrants the first modification to the WHILE language and its semantics: assignments are allowed in Boolean expressions. The second modification is that the Boolean OR function will be replaced by a short-circuit version:

$$B ::= \top \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg B \mid B_1 \mathbin{^\circ\!\vee} B_2 \mid v := a$$

The new semantics for Boolean expressions are like the semantics defined by Van Eijck, with as major difference that there are now semantics defined for assignments:

$$[\![v := a]\!]_g := T$$

Furthermore, Boolean expressions now might introduce a state change, so every command containing a Boolean expression (which for now only is the IF THEN ELSE command) should account for that. In structural operational semantics, we take a look at how the Boolean expression changes the state and perform the remaining actions in that new state:

$$\frac{(g, B) \Longrightarrow g'}{(g, \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) \Longrightarrow (g', C_1)} [\![B]\!]_g = T$$

And similar for the case that $[\![B]\!]_g = F$.

As said, there is one more thing that needs to be modified in the language above. In order to be properly able to reason about side effects, the order in which the tests get executed is important. Because of that, the OR construct in Boolean expressions needs to be replaced by a short-circuit directed version:

$$[\![B_1 \mathbin{\overset{\circ}{\vee}} B_2]\!]_g := \begin{cases} T \text{ if } [\![B_1]\!]_g = T \\ T \text{ if } [\![B_1]\!]_g = F \text{ and for } (g, B) \Longrightarrow g', [\![B_2]\!]_{g'} = T \\ F \text{ otherwise} \end{cases}$$

We will make use of its dual, the short-circuit left and ($\mathbin{\overset{\circ}{\wedge}}$) too. It is defined similarly as above. As a convention, from here on $\mathbin{\overset{\circ}{\vee}}$ and $\mathbin{\overset{\circ}{\wedge}}$ can be used interchangeably in definitions, unless explicitly stated otherwise. Both $\mathbin{\overset{\circ}{\vee}}$ as well as $\mathbin{\overset{\circ}{\wedge}}$ are associative. We again omit brackets wherever possible.

All we have left to define now is the state change a Boolean can cause. This is defined as follows:

$$(g, B) \Longrightarrow \begin{cases} g[v \mapsto [\![t]\!]_g] & \text{if } B = (v := t) \\ g & \text{o.w.} \end{cases}$$

Missing in the above WHILE language are the random assignment and the existential quantifier. This is because I have decided to drop them. The reason for that is that they can cause non-deterministic behavior and in this thesis, we are not interested in the (side effects of) non-deterministic programs. In fact it is questionable whether we can say anything about side effects in non-deterministic programs, but I will return to that in my possibilities for future work in Chapter 9. Aside from that, in our context of (imperative) programs, the random assignment is an unusual concept at best. The same goes for the formula $\exists v \phi$.

With those modifications to the toy language in mind, we can take a look at the similar modifications that need to be made to QDL. In the resulting dynamic logic $\mathrm{DLA_f}$, we keep the same terms:

$$t ::= v \mid f t_1 \dots t_n$$

In $\mathrm{DLA_f}$ we of course drop the random assignment and existential quantifier, too. By dropping them, we lose the quantified character of QDL. Because of that, the resulting logic is no longer called a quantified dynamic logic. The first major change to QDL, besides the absence of the random assignment and the existential quantifier, is that I replace the $\langle \pi \rangle \phi$ command with the weaker $[v := t]\top$:

$$\phi ::= \top \mid R t_1 \dots t_n \mid t_1 = t_2 \mid \neg \phi \mid \phi_1 \mathbin{\overset{\circ}{\vee}} \phi_2 \mid \phi_1 \mathbin{\overset{\circ}{\wedge}} \phi_2 \mid [v := t]\top$$

This modification explicitly expresses the possibility of assignments in formulas. All other programs, however, are no longer allowed in formulas. Because of this modification we will avoid a number of problems that QDL has, while keeping the desired functionality that there should be room for assignments in formulas. I will address these problems in detail in Section 3.3.

We have also replaced the $\vee$ connective with its short-circuit variant ($\mathbin{\overset{\circ}{\vee}}$) and for convenience, have explicitly introduced its dual ($\mathbin{\overset{\circ}{\wedge}}$). We will return to the motivation for this change at the end of this chapter.

We also need to replace the QDL-formula associated with this command (QDL9). The truth in M for the new command is defined as follows:

$$M \models_g [v := t]\top \text{ always} \tag{DLA9}$$

It should come as no surprise that this always succeeds, since assignments always succeed and yield `true`. Since this formula always succeeds, we replaced the possibility modality ($\langle v := t\rangle\top$) for the necessity modality ($[v := t]\top$). The reason we keep this formula in the form of a modality at all (and not just $v := t$), is because formulas of this form can change the initial valuation. This is in sharp contrast to the basic formulas $t_1 = t_2$ and $Rt_1 \dots t_2$, which do not change the initial valuation and are typically not modalities. Because of that, it is unintuitive to write the assignment formula as $v := t$.

On a side note: in our toy language we *do* simply write $v := t$ for the assignment, regardless of where it occurs. This is because in the world of (imperative) programming, assignments are allowed in steering fragments.

We will see below that we are going to accept possible state changes in formulas, in contrast to the original QDL versions. For this we will use a mechanism to determine when a state change happens, that is, a function that returns the program(s) that are encountered when evaluating a formula ϕ. This function is defined as follows:

Definition 1. *The **program extraction function** $\Pi_g^M : \phi \to \pi$ returns for formula ϕ the program(s) that are encountered when evaluating the formula given modal M and initial valuation g. It is defined recursively as follows:*

$$\begin{aligned}
\Pi_g^M(\top) &= ?\top \\
\Pi_g^M(Rt_1 \dots t_n) &= ?\top \\
\Pi_g^M(t_1 = t_2) &= ?\top \\
\Pi_g^M(\neg\phi) &= \Pi_g^M(\phi) \\
\Pi_g^M(\phi_1 \mathbin{\overset{\circ}{\vee}} \phi_2) &= \begin{cases} \Pi_g^M(\phi_1) & \text{if } M \models_g \phi_1 \\ \Pi_g^M(\phi_1); \Pi_h^M(\phi_2) & \text{if } M \not\models_g \phi_1 \text{ and } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M \end{cases} \\
\Pi_g^M(\phi_1 \mathbin{\overset{\circ}{\wedge}} \phi_2) &= \begin{cases} \Pi_g^M(\phi_1) & \text{if } M \not\models_g \phi_1 \\ \Pi_g^M(\phi_1); \Pi_h^M(\phi_2) & \text{if } M \models_g \phi_1 \text{ and } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M \end{cases} \\
\Pi_g^M([v := t]\top) &= (v := t)
\end{aligned}$$

In the first three cases, no programs are encountered. Therefore, the program extraction function returns the empty program ($?\top$). The formula $\neg\phi$ is transparent, that is, it returns any program encountered in its subformula ϕ. Because of the short-circuit character of $\overset{\circ}{\vee}$ and $\overset{\circ}{\wedge}$, a case distinction is made here: in case of $\overset{\circ}{\vee}$, ϕ_2 will not be evaluated if ϕ_1 yields true, therefore only the program(s) encountered in ϕ_1 will be returned. Otherwise, the result is a concatenation of the program(s) encountered in ϕ_1 and ϕ_2. Obviously, for $\overset{\circ}{\wedge}$ the opposite is the case and this clause is derivable from the previous one using duality. Finally, if the formula is an assignment, the program equivalent of that assignment is returned.

Because the evaluation of a formula now can cause a state change, the original definition for the truth in M of $\overset{\circ}{\vee}$ (QDL7) is no longer valid. In case ϕ_1

contains an assignment, ϕ_2 must be evaluated in a different valuation, namely the one resulting after evaluating ϕ_1 in the initial valuation:

$$M \models_g \phi_1 \mathbin{\overset{\circ}{\vee}} \phi_2 \text{ iff for } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M, M \models_g \phi_1 \text{ or } M \models_h \phi_2 \qquad \text{(DLA7a)}$$

Since we have added $\mathbin{\overset{\circ}{\wedge}}$ to formulas as well, we also explicitly have to define the truth in M for $\mathbin{\overset{\circ}{\wedge}}$, which is similar to the updated definition of $\mathbin{\overset{\circ}{\vee}}$:

$$M \models_g \phi_1 \mathbin{\overset{\circ}{\wedge}} \phi_2 \text{ iff for } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M, M \models_g \phi_1 \text{ and } M \models_h \phi_2 \qquad \text{(DLA7b)}$$

Although $\mathbin{\overset{\circ}{\vee}}$ and $\mathbin{\overset{\circ}{\wedge}}$ use short-circuit evaluation, we do not explicitly have to define them as such above because we will make sure, via the program extraction function and an updated version of QDL11 (see below), that the valuation does not change as a result of ϕ_2 when $M \models_g \phi_1$ is true (in case of $\mathbin{\overset{\circ}{\vee}}$) or false (in case of $\mathbin{\overset{\circ}{\wedge}}$).

We can now turn our attention to programs in $\mathrm{DLA_f}$. Besides the absence of the random assignment, what a program π can be does not change:

$$\pi ::= v := t \mid ?\phi \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^*$$

To remedy the problem that more things can be expressed in QDL than there are semantics for, we need, as mentioned earlier, to accept that a state change can occur when evaluating a program containing formulas. In the case of QDL, that only is the test instruction, given semantics earlier in QDL11. So, as second major change we need to replace QDL11 by:

$${}_g[\![?\phi]\!]_h^M \text{ iff } \begin{cases} M \models_g \phi \text{ and } g = h & \text{if } \Pi_g^M(\phi) = ?\top \\ M \models_g \phi \text{ and } {}_g[\![\Pi_g^M(\phi)]\!]_h & \text{otherwise} \end{cases} \qquad \text{(DLA11)}$$

The choice here is in place to avoid looping behavior when evaluating ${}_g[\![?\top]\!]_h$.

The definitions above make extensive use of the empty program ($?\top$). In what follows, it will be handy to know that the empty program is truly empty. In particular, we would like to have $\pi; ?\top = \pi$ and $?\top; \pi = \pi$. I will prove that below.

Lemma 3.1.1. *For any program π, initial valuation g, output valuation h and model M*

$${}_g[\![\pi; ?\top]\!]_h^M \textit{ iff } {}_g[\![\pi]\!]_h^M$$

Proof. The proof follows from the above defined QDL-axioms:

$${}_g[\![\pi; ?\top]\!]_h^M \text{ iff } \exists f \ {}_g[\![\pi]\!]_f^M \text{ and } {}_f[\![?\top]\!]_h$$

Since we have ${}_f[\![?\top]\!]_h$ iff $f = h$ and $M \models_f \top$, and since the latter is always true, we have

$${}_g[\![\pi; ?\top]\!]_h^M \text{ iff } {}_g[\![\pi]\!]_h^M$$

□

Lemma 3.1.2. *For any program π, initial valuation g, output valuation h and model M*

$${}_g[\![?\top; \pi]\!]_h^M \textit{ iff } {}_g[\![\pi]\!]_h^M$$

Proof. Similar as for Lemma 3.1.1. □

The change to QDL11 has remedied the problem that there are expressions in QDL for which there are no semantics defined. Of course I made a second major change — namely replacing $\langle\pi\rangle\phi$ by $[v := t]\top$. The reason for that will come to light as soon as I will reintroduce the WHILE command in Section 3.3. Before I will do that, however, I will first discuss a working example to provide some more insight into the inner workings of $\mathrm{DLA_f}$.

3.2 A working example

In this section I will present a working example to illustrate how $\mathrm{DLA_f}$ works. I will use the following program, presented here in our toy language:

$x := 1;$
IF $(x := x + 1 \mathbin{\diamond\!\!\wedge} x = 2)$
THEN $y := 1$
ELSE $y := 2$

In $\mathrm{DLA_f}$, this translates to:

$x := 1;$
$(?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} x = 2); y := 1)$
$\cup$
$(?\neg([x := x + 1]\top \mathbin{\diamond\!\!\wedge} x = 2); y := 2)$

The valuations $g, h, \ldots$ are defined for all variables $v \in \mathcal{V}$, i.e. they are total functions. Usually we are only interested in a small number of variables, e.g. x and y, in which case we talk about a valuation g such that $g(x) = [\![t]\!]^M_g, g(y) = [\![t']\!]^M_g$, or if valuation h is an update of valuation g, $h = g[x \mapsto [\![t]\!]^M_g, y \mapsto [\![t']\!]^M_g]$ (which is a shorthand for $g[x \mapsto [\![t]\!]^M_g][y \mapsto [\![t']\!]^M_g]$). In all examples we discuss we take for M the model of the natural numbers and we use numerals to denote its elements.

Since we are working on natural numbers, as constants we have n ranging over numerals, as functions we have $+, *$ and $\dot{-}$, and as extra relation we have $\leq$. Our model M contains those constants, functions and relations. Assume we have an initial valuation g that sets x and y to 0: $g(x) = g(y) = 0$. We will now first show how the program in our toy language gets evaluated using the structural operational semantics we provided in Chapter 2:

$$(g, \big(x := 1; \text{IF } (x := x + 1 \mathbin{\diamond\!\!\wedge} x = 2) \text{ THEN } y := 1 \text{ ELSE } y := 2\big)) \Longrightarrow$$
$$(g[x \mapsto 1], \big(\text{IF } (x := x + 1 \mathbin{\diamond\!\!\wedge} x = 2) \text{ THEN } y := 1 \text{ ELSE } y := 2\big))$$

We now need to know if $[\![(x := x + 1 \mathbin{\diamond\!\!\wedge} x = 2)]\!]_{g[x\mapsto 1]} = T$. We can easily see that it is and furthermore updates the valuation again by incrementing x by 1. Thus we get as valuation $g[x \mapsto 2]$ and we can finish our evaluation as follows:

$$(g[x \mapsto 2], \big(y := 1\big)) \Longrightarrow g[x \mapsto 2, y \mapsto 1]$$

Having seen how our example program evaluates using the semantics for our toy language, we can turn our attention to the evaluation using $\mathrm{DLA_f}$. We need to ask ourselves if ${}_g[\![\pi]\!]^M_h$ exists (with π the program above), that is, if there is a valuation h that models the state of the program after being executed on initial valuation g.

Schematically, π can be broken down as follows:

$$\begin{aligned}
\pi &::= \pi_0; \pi_1 \\
\pi_0 &::= x := 1 \\
\pi_1 &::= (?\phi_0; \pi_2) \cup (?\neg\phi_0; \pi_3) \\
\pi_2 &::= y := 1 \\
\pi_3 &::= y := 2 \\
\phi_0 &::= \phi_1 \mathbin{\diamondsuit\!\!\wedge} \phi_2 \\
\phi_1 &::= [x := x + 1]\top \\
\phi_2 &::= x = 2
\end{aligned}$$

The break-down above paves the way to evaluate ${}_g[\![\pi]\!]^M_h$ using the $\mathrm{DLA_f}$-axioms given in the previous sections. We start by applying QDL12:

$$\begin{aligned}
{}_g[\![\pi]\!]^M_h = {}_g[\![\pi_0; \pi_1]\!]^M_h \\
\text{iff } \exists f \text{ s.th. } {}_g[\![\pi_0]\!]^M_f \text{ and } {}_f[\![\pi_1]\!]^M_h
\end{aligned}$$

We find f by evaluating ${}_g[\![x := 1]\!]^M_f$ using QDL10 and QDL1:

$$\begin{aligned}
{}_g[\![x := 1]\!]^M_f \text{ iff } f &= g[x \mapsto [\![1]\!]^M_g] \\
&= g[x \mapsto 1]
\end{aligned}$$

Now we need to evaluate ${}_f[\![(?\phi_0; \pi_2) \cup (?\neg\phi_0; \pi_3)]\!]^M_h$. Using QDL13, we get:

$${}_f[\![(?\phi_0; \pi_2) \cup (?\neg\phi_0; \pi_3)]\!]^M_h \text{ iff } {}_f[\![?\phi_0; \pi_2]\!]^M_h \text{ or } {}_f[\![?\neg\phi_0; \pi_3]\!]^M_h$$

First we turn our attention to ${}_f[\![?\phi_0; \pi_2]\!]^M_h$. Using QDL12 again we get $\exists d$ such that ${}_f[\![?\phi_0]\!]^M_d$ and ${}_d[\![\pi_2]\!]^M_h$. To evaluate the former, we need to use our own rule DLA11. Here we need the program extraction function Π for the first time:

$$\begin{aligned}
{}_f[\![?\phi_0]\!]^M_d = {}_f[\![?([x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2))]\!]^M_d \\
\text{iff } M \models_f [x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2) \\
\text{and } {}_f[\![\Pi^M_f([x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2))]\!]^M_d
\end{aligned}$$

We will first have a look at the program extraction function Π. Below we will see how it calculates the programs that are encountered while evaluating the formula $(x := x + 1) \mathbin{\diamondsuit\!\!\wedge} (x = 2)$:

$$\begin{aligned}
\Pi^M_f([x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2)) &= \Pi^M_f([x := x + 1]\top); \Pi^M_f(x = 2) \\
&= (x := x + 1); ?\top
\end{aligned}$$

Therefore, we have:

$$\begin{aligned}
{}_f[\![?\phi_0]\!]^M_d = {}_f[\![?([x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2))]\!]^M_d \\
\text{iff } M \models_f [x := x + 1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2) \\
\text{and } {}_f[\![x := x + 1; ?\top]\!]^M_d \text{ iff } {}_f[\![x := x + 1]\!]^M_d
\end{aligned}$$

The first of these two, $M \models_f [x := x+1]\top \,{}_{\diamond}\!\wedge\, (x = 2)$, nicely shows why we need an updated version of ${}_{\diamond}\!\wedge$ and ${}^{\diamond}\!\vee$. As we already noticed the test ϕ_0 contains a program (the assignment $x := x+1$) and therefore the state (valuation) changes. As we will see, this will change the outcome of the second part of the test. We need DLA7b and our program extraction function Π here:

$$M \models_f (x := x+1) \,{}_{\diamond}\!\wedge\, (x = 2) \text{ iff for } {}_f[\![x := x+1]\!]^M_c, M \models_f (x := x+1) \text{ and } M \models_c (x = 2)$$

$M \models_f (x := x+1)$ is defined by DLA8 to be always true. Applying QDL10 on ${}_f[\![x := x+1]\!]^M_c$ will give us $c = f[x \mapsto 2]$. We can then apply QDL5 on $M \models_c (x = 2)$:

$$M \models_c (x = 2) \text{ iff } [\![x]\!]^M_c = [\![2]\!]^M_c$$

We can easily see (using QDL1) that $[\![x]\!]^M_c = c(x) = 2 = [\![2]\!]^M_c$. Therefore, we have $M \models_c (x = 2)$ and thus $M \models_f [x := x+1]\top \,{}_{\diamond}\!\wedge\, (x = 2)$.

We now need to finish the evaluation of DLA11 by evaluating ${}_f[\![x := x+1]\!]^M_d$. This can again be done using QDL10 and gives us $d = f[x \mapsto 2]$. Because the test ϕ_0 has now succeeded, we can continue to the evaluation of ${}_d[\![\pi_2]\!]^M_h = {}_d[\![y := 1]\!]^M_h$. This will give us $h = d[y \mapsto 1]$. Having already established that $?\phi_0$ succeeds, we also know that $?\neg\phi$ will not succeed. Therefore, we are done with the evaluation of this program π, getting that ${}_g[\![\pi]\!]^M_h$ with $g(x) = g(y) = 0$ is indeed possible with $h = g[x \mapsto 2, y \mapsto 1]$.

3.3 Re-introducing WHILE

In Section 2.2 I introduced our toy language, which was like Van Eijck's WHILE language, but without a WHILE (or: guarded iteration) programming command. Now that we have seen $\mathrm{DLA_f}$ in action in our simplified toy language, it is time to re-introduce the WHILE command. After doing that, we will see that the re-introduction of WHILE raises some more issues that warrant the second modification I made to QDL, namely replacing the formula $\langle\pi\rangle\phi$ with $[v := t]\top$.

3.3.1 The WHILE command

The WHILE command takes the form WHILE B DO C. The complete list of programming commands in our toy language then is:

$$C ::= \text{SKIP} \mid \text{ABORT} \mid v := a \mid C_1; C_2 \mid \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \mid \text{WHILE } B \text{ DO } C$$

In structural operational semantics, the semantics for the guarded iteration are as follows. There are two options: if the guard (B) is not satisfied, command C is not executed. Instead, the command finishes, with as only (possible) change the change that the evaluation of guard B has made to the state:

$$\frac{(g, B) \Longrightarrow g'}{(g, \text{WHILE } B \text{ DO } C) \Longrightarrow g'} [\![B]\!]_g = F$$

If the guard *is* satisfied, the rule becomes a little more complicated because command C gets executed in a state which is possibly changed by guard B.

Like before, we have two cases: one for which C finishes in a single step and one for which it does not.

$$\frac{(g, B) \Longrightarrow g' \qquad (g', C) \Longrightarrow g''}{(g, \text{WHILE } B \text{ DO } C) \Longrightarrow (g'', \text{WHILE } B \text{ DO } C)} [\![B]\!]_g = T$$

$$\frac{(g, B) \Longrightarrow g' \qquad (g', C) \Longrightarrow (g'', C')}{(g, \text{WHILE } B \text{ DO } C) \Longrightarrow (g'', C'; \text{WHILE } B \text{ DO } C)} [\![B]\!]_g = T$$

3.3.2 WHILE in $\mathbf{DLA_f}$

In PDL, and therefore QDL and DLA_f, WHILE is expressed as follows:

$$\text{WHILE } \phi \text{ DO } \alpha := (?\phi; \alpha)^*; ?\neg\phi$$

Thanks to the updated rule for $?\phi$ (DLA11), DLA_f is able to handle programs with WHILE perfectly. To see how this works, consider the following example:

$x := 0;$
$y := 0;$
WHILE $(x := x + 1 \curlywedge x \leq 2)$
DO $y := y + 1$

In DLA_f, this translates to:

$x := 0;$
$y := 0;$
$(?([x := x + 1]\top \curlywedge x \leq 2); y := y + 1)^*;$
$?\neg([x := x + 1]\top \curlywedge x \leq 2)$

After the first two commands, we have $g(x) = g(y) = 0$. We now need to look at how the * operator is evaluated. QDL14 states that ${}_g[\![\pi^*]\!]^M_h$ iff $g = h$ or ${}_g[\![\pi; \pi^*]\!]^M_h$). This means that π is either executed not at all (in which case $g = h$) or at least once. In our case, $\pi = ?([x := x + 1]\top \curlywedge x \leq 2); y := y + 1$.

The first option is that π is executed not at all, in which case $g = h$. However, under this valuation h there is no possible valuation h' after evaluation of the next program command $(?\neg([x := x+1]\top \curlywedge x \leq 2))$. In other words, ${}_h[\![?\neg([x := x + 1]\top \curlywedge x \leq 2)]\!]^M_{h'}$ is false. Therefore, we have to turn our attention to the other option given by the * command, which is ${}_g[\![\pi; \pi^*]\!]^M_h$. For the evaluation of this we first need QDL12, which tells us that there has to be an f such that ${}_g[\![\pi]\!]^M_f$ and ${}_f[\![\pi^*]\!]^M_h$. In Section 3.2 we have already seen how ${}_g[\![\pi]\!]^M_f$ evaluates; it will succeed and result in a new valuation $f = g[x \mapsto 1, y \mapsto 1]$.

Now we need to evaluate π^* again, but this time with a different initial valuation (namely f). This loop continues until we arrive at a valuation f' for which the final program command (the test $?\neg([x := x + 1]\top \curlywedge x \leq 2)$) *will* succeed. In our example, this happens in the second iteration, when we have $f' = g[x \mapsto 2, y \mapsto 2]$, giving us a resulting valuation $h = g[x \mapsto 3, y \mapsto 2]$, which is exactly what we would expect given this WHILE loop.

3.3.3 Looping behavior and abnormal termination

An interesting problem regarding the WHILE language and QDL is that WHILE T DO SKIP (looping behavior) and ABORT (abnormal termination) are indistinguishable. In some semantics, such as natural semantics, this is also the case [11]. In structural operational semantics, however, there is an (infinite) derivation sequence for WHILE T DO SKIP, whereas there is no derivation sequence for ABORT.

Using the standard lemma that $\langle \pi_1; \pi_2 \rangle \phi \leftrightarrow \langle \pi_1 \rangle \langle \pi_2 \rangle \phi$ (cf. [15, 11]) we can prove the equivalence of WHILE T DO SKIP and ABORT in QDL. To do so, we need to ask if $\langle (?\top; ?\top)^*; ?\bot \rangle \phi \leftrightarrow \langle ?\bot \rangle \phi$.

Theorem 3.3.1. *In QDL, looping behavior and abnormal termination are equivalent: for any* ϕ

$$\langle (?\top; ?\top)^*; ?\bot \rangle \phi \leftrightarrow \langle ?\bot \rangle \phi$$

Proof. We will work out the left part first:

$$\langle (?\top; ?\top)^*; ?\bot \rangle \phi \leftrightarrow \langle (?\top; ?\top)^* \rangle \langle ?\bot \rangle \phi$$

So we have $\langle (?\top; ?\top)^* \rangle \psi$ with $\psi = \langle ?\bot \rangle \phi$. Truth of the former in a random model M and for an initial valuation g is defined as follows:

$$M \models_g \langle (?\top; ?\top)^* \rangle \psi \text{ iff for some } h \text{ with } {}_g[\![(?\top; ?\top)^*]\!]^M_h, M \models_h \psi$$

Furthermore we have

$${}_g[\![(?\top; ?\top)^*]\!]^M_h \text{ iff } g = h \text{ or } {}_g[\![(?\top; ?\top); (?\top; ?\top)^*]\!]^M_h$$

We have seen in the previous section how such a formula evaluates; after one iteration we will have ${}_g[\![?\top; ?\top]\!]^M_f$, with $f = h$, as one of the options the * command gives us. Finally we have

$$\begin{aligned} {}_g[\![?\top; ?\top]\!]^M_h &= {}_g[\![?\top]\!]^M_h \\ &\text{iff } g = h \text{ and } M \models_g \top \end{aligned}$$

This is always the case, so indeed there is an h such that ${}_g[\![(?\top; ?\top)^*]\!]^M_h$ (namely $h = g$). Therefore, determining the truth of $M \models_g \langle (?\top; ?\top)^* \rangle \psi$ comes down to determining the truth of $M \models_g \psi$, which is $M \models_g \langle ?\bot \rangle \phi$.

Since that is exactly the right hand side of the equation we started out with, we indeed have that

$$\langle (?\top; ?\top)^*; ?\bot \rangle \phi \leftrightarrow \langle ?\bot \rangle \phi$$

□

Not being able to distinguish between looping behavior and abnormal termination seems undesirable. It is because of this that I have decided to drop the $\langle \pi \rangle \phi$ formulas and replace it by the weaker, but less problematic formulas $[v := t]\top$. Looping behaviour can now no longer be proven to be equivalent to abnormal termination. Furthermore, we avoid problems with formulas that require infinite evaluations, such as $\langle (?\top)^*; ?\bot \rangle \phi$.

Because looping behavior and abnormal termination can no longer be proven equal in $\mathrm{DLA_f}$, the relational meaning of $\mathrm{DLA_f}$-instructions now is an instance

of the structural operational semantics we defined for our toy language, with the valuations as 'states'. Naturally, this is what we want, since it expresses that $\mathrm{DLA_f}$ is a fully defined system that has the behavior we would expect given our toy language.

This modification also underlines the usefulness of the switch to short-circuit versions of the logical connectives ($\overset{\circ}{\vee}$ and its dual $\overset{\circ}{\wedge}$). In QDL, the steering fragment of the program

$$\text{IF } x := x + 1 \text{ AND } x == 2 \text{ THEN } a \text{ ELSE } b$$

can be expressed using $?(\langle x := x + 1\rangle(x = 2))$. In $\mathrm{DLA_f}$ such an expression now no longer is allowed. However, having $\overset{\circ}{\wedge}$ and $\overset{\circ}{\vee}$ in $\mathrm{DLA_f}$ allows us to provide a perhaps even more natural translation of this program, namely $?([x := x+1]\top \overset{\circ}{\wedge} x = 2)$. The full evaluation versions of these logical connectives ($\wedge$ and $\vee$) would not do, because the order of the program instructions is important here. As we will see in Chapter 4, we do not *need* $\overset{\circ}{\wedge}$ and $\overset{\circ}{\vee}$ in $\mathrm{DLA_f}$, but the fact they provide natural translations of this kind, together with the fact that having logical connectives defined is standard in dynamic logic, is reason enough to keep them.

4 Terminology

In this chapter I will present the terminology I will be using in the remainder of this thesis. In particular, I will present a more fine-grained breakdown of the definitions for formulas, instructions and programs. Furthermore, I will introduce a property of formulas called normal form and use that to prove yet another property of $\mathrm{DLA_f}$ regarding complex steering fragments. Next, I will introduce a subclass of programs called deterministic programs. Finally, I will introduce a property of deterministic programs called canonical form.

4.1 Formulas, instructions and programs

In this section I will present the more fine-grained breakdown of the definitions for formulas, instructions and programs.

Definition 2. ***Formulas*** *can either be primitive or compound formulas.* ***Primitive formulas*** *are written as φ and defined as follows:*

$$\varphi ::= \top \mid Rt_1 \dots t_n \mid t_1 = t_2 \mid [v := t]\top$$

Compound formulas *are written as ϕ and defined similarly, but with negation and short-circuit disjunction and conjunction as addition:*

$$\phi ::= \top \mid Rt_1 \dots t_n \mid t_1 = t_2 \mid \neg\phi \mid \phi_1 \mathbin{\overset{\circ}{\vee}} \phi_2 \mid \phi_1 \mathbin{\overset{\circ}{\wedge}} \phi_2 \mid [v := t]\top$$

Definition 3. ***Instructions*** *can either be single instructions or basic instructions.* ***Single instructions*** *are written as ρ and defined as follows:*

$$\rho ::= (v := t) \mid ?\varphi$$

Basic instructions *are written as ϖ and have a little less restrictive definition regarding tests:*

$$\varpi ::= (v := t) \mid ?\phi$$

This means that single instructions form a subset of basic instructions:

$$\rho \subseteq \varpi$$

Definition 4. ***Programs*** *are written as π and consist of one or more basic instructions joined by either concatenation (;), union ($\cup$) or repetition (*):*

$$\pi ::= \varpi \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^*$$

4.2 Normal forms of formulas

In this section I will introduce a property of formulas called normal form and use that to prove a property of $\mathrm{DLA_f}$ regarding complex steering fragments. I will start with the former.

Definition 5. *A formula is said to be in its* ***normal form*** *iff all negations (if any) that occur in the formula are on atomic level, that is if the negations only have primitive formulas as their argument (i.e. are of the form $\neg\varphi$).*

Proposition 1. *Any formula can be rewritten into its normal form such that its relational meaning is preserved.*

Proof. Left-sequential versions of De Morgan's laws are valid for formulas (we come back to this point in Chapter 5): given model M and initial valuation g we prove that

$$M \models_g \neg(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2) \iff M \models_g \neg\phi_1 \mathbin{\circ\!\!\vee} \neg\phi_2$$

For $\Longrightarrow$, first assume that $M \models_g \phi_1$, thus $M \not\models_h \phi_2$ for ${}_g[\![\Pi_g^M(\phi_1)]\!]_h^M$, thus $M \models_h \neg\phi_2$, and thus $M \models_g \neg\phi_1 \mathbin{\circ\!\!\vee} \neg\phi_2$. If $M \not\models_g \phi_1$, then $M \models_g \neg\phi_1$, and thus also $M \models_g \neg\phi_1 \mathbin{\circ\!\!\vee} \neg\phi_2$.

In order to show $\Longleftarrow$, first assume that $M \models_g \neg\phi_1$, thus $M \not\models_g \phi_1 \mathbin{\diamond\!\!\wedge} \phi_2$, thus $M \models_g \neg(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2)$. If $M \models_g \phi_1$, then $M \models_h \neg\phi_2$ for ${}_g[\![\Pi_g^M(\neg\phi_1)]\!]_h^M$, so $M \not\models_g \phi_1 \mathbin{\diamond\!\!\wedge} \phi_2$, and thus $M \models_g \neg(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2)$.

The dual statement can also easily be proved. □

The set of side effects caused by the evaluation of a formula does not change under rewritings of this kind. Using normal forms, we can derive an interesting property of $\mathrm{DLA_f}$:

Proposition 2. *Let ϕ be a formula. The program $?\phi$ can be rewritten to a form in which only primitive formulas or negations thereof occur in tests, such that its relational meaning is preserved.*

Proof. Let ϕ_n be a normal form of ϕ and assume ϕ_n is not a primitive formula or the negation thereof. Then, ϕ_n either is of the form $\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2$ or $\phi_1 \mathbin{\circ\!\!\vee} \phi_2$. For conjunctions, it is easy to see that the program $?\phi$ can be rewritten as meant in the proposition:

$$?(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2) = ?\phi_1; ?\phi_2$$

We can assume by induction that ϕ_1 and ϕ_2 has been rewritten into a form in which only primitive formulas and negations occur, too. We now need to prove that these programs have the same relational meaning, that is given model M and initial valuation g

$${}_g[\![?(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2)]\!]_h^M \text{ iff } {}_g[\![?\phi_1; ?\phi_2]\!]_h^M$$

If $M \not\models_g \phi_1$, then h does not exist in both cases. If, for ${}_g[\![\Pi_g^M(\phi_1)]\!]_f^M$, $M \not\models_f \phi_2$, h does not exist in both cases either. Otherwise, on the left hand side, we get h by applying DLA11:

$$ {}_g[\![\Pi_g^M(\phi_1 \mathbin{\diamond\!\!\wedge} \phi_2)]\!]_h^M $$

which by definition of the program extraction function, since $M \models_g \phi_1$, equals

$$ {}_g[\![\Pi_g^M(\phi_1); \Pi_f^M(\phi_2)]\!]_h^M $$

On the right hand side, we get h by first applying QDL12, then applying DLA11 twice and finally applying QDL12 again:

$$\begin{aligned} {}_g[\![?\phi_1; ?\phi_2]\!]_h^M &\text{ iff } \exists f \text{ s.th. } {}_g[\![?\phi_1]\!]_f^M \text{ and } {}_f[\![?\phi_2]\!]_h^M \\ &\text{iff } \exists f \text{ s.th. } {}_g[\![\Pi_g^M(\phi_1)]\!]_f^M \text{ and } {}_f[\![\Pi_f^M(\phi_2)]\!]_h^M \\ &\text{iff } {}_g[\![\Pi_g^M(\phi_1); \Pi_f^M(\phi_2)]\!]_h^M \end{aligned}$$

For disjunctions, the rewritten version is slightly more complex:

$$?(\phi_1 \mathbin{\diamond\!\!\vee} \phi_2) = ?\phi_1 \cup ?\neg\phi_1; ?\phi_2 $$

We can prove that given model M and initial valuation g

$$ {}_g[\![?(\phi_1 \mathbin{\diamond\!\!\vee} \phi_2)]\!]_h^M \text{ iff } {}_g[\![?\phi_1 \cup ?\neg\phi_1; ?\phi_2]\!]_h^M $$

in a similar fashion as above. If $M \models_g \phi_1$, then in both cases h is obtained by

$$ {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M $$

If $M \not\models_g \phi_1$, then if for ${}_g[\![\Pi_g^M(\phi_1)]\!]_f^M$, $M \not\models_f \phi_2$, in both cases h does not exist. If $M \models_f \phi_2$, then on the left hand side h is obtained via

$$ {}_g[\![\Pi_g^M(\phi_1 \mathbin{\diamond\!\!\vee} \phi_2)]\!]_h^M =_g [\![\Pi_g^M(\phi_1); \Pi_f^M(\phi_2)]\!]_h^M $$

And on the right hand side, h is obtained by

$$\begin{aligned} {}_g[\![?\neg\phi_1; ?\phi_2]\!]_h^M &\text{ iff } \exists f \text{s.th.}_g[\![?\neg\phi_1]\!]_f^M \text{ and } {}_f[\![?\phi_2]\!]_h^M \\ &\text{iff } \exists f \text{s.th.}_g[\![\Pi_g^M(\phi_1)]\!]_f^M \text{ and } {}_f[\![\Pi_f^M(\phi_2)]\!]_h^M \\ &\text{iff } {}_g[\![\Pi_g^M(\phi_1); \Pi_f^M(\phi_2)]\!]_h^M \end{aligned}$$

□

On a side note, a similar result can be obtained for QDL. Here the program $?(\phi_1 \vee \phi_2)$ can be rewritten to

$$ (?\phi_1; ?\phi_2) \cup (?\phi_1; ?\neg\phi_2) \cup (?\neg\phi_1; ?\phi_2) $$

The differences between the $\mathrm{DLA_f}$ version of the same rule are there because QDL uses full evaluation. Therefore, ϕ_2 has to be evaluated even when ϕ_1 is true, although ϕ_2 does not have to be true anymore.

4.3 Deterministic programs and canonical forms

Defining side effects for entire programs can be complicated. This is because two composition operators, namely union and repetition, can be non-deterministic. We are, however, not interested in (the side effects of) non-deterministic programs, even though they can be expressed in $\mathrm{DLA_f}$.[1] To be exact, we are only interested in *if ... then ... else* constructions and *while* constructions, which in $\mathrm{DLA_f}$ are expressed as follows:

$$\text{IF } \phi \text{ THEN } \pi_1 \text{ ELSE } \pi_2 := (?\phi;\pi_1) \cup (?\neg\phi;\pi_2)$$
$$\text{WHILE } \phi \text{ DO } \pi := (?\phi;\pi)^*;?\neg\phi$$

To formally specify this, we introduce *deterministic programs*, which cf. [14, 11] are defined as follows:

Definition 6. *A **deterministic program** $d\pi$ is a DLA_f-program in one of the following forms:*

$$d\pi ::= \varpi \mid \ d\pi_1;d\pi_2 \mid (?\phi;d\pi_1) \cup (?\neg\phi;d\pi_2) \mid ((?\phi;d\pi)^*;?\neg\phi)$$

There are two interesting properties of deterministic programs. The first is regarding programs of the form $(?\phi;\pi)^*;?\neg\phi$. In this case there will only ever be exactly one situation in which the program gets evaluated.[2] After all, there is exactly one repetition loop for which the test $?\phi$ succeeds, but will fail the next time it is evaluated. We can formalize this intuition in the following proposition:

Proposition 3. *Let $d\pi = (?\phi;d\pi_0)^*;?\neg\phi$ be a deterministic program. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$. There is a unique $n \in \mathbb{N}_0$ such that*

$${}_g[\![d\pi]\!]^M_h \text{ iff } {}_g[\![(?\phi;d\pi_0)^n;?\neg\phi]\!]^M_h$$

where $(d\pi_1)^0;d\pi_2 = d\pi_2$ and $(d\pi_1)^{n+1};d\pi_2 = d\pi_1;(d\pi_1)^n;d\pi_2$.

Proof. We first prove that there is at least one $n \in \mathbb{N}_0$ for which the above equation holds. Assume such an n does not exist. This means that $?\neg\phi$ can never be evaluated, which is a contradiction with our requirement that there is a valuation h such that ${}_g[\![d\pi]\!]^M_h$.

Next, we have to prove that there is at most one such n. Let g_i be the valuation such that ${}_g[\![(?\phi;d\pi_0)^i]\!]^M_{g_i}$. By writing this out and then applying DLA11, we know that for $i < n$, we have $M \models_{g_i} ?\phi$. Therefore, for valuation g_i with $i < n$ we cannot evaluate $?\neg\phi$ and thus there is no $i < n$ for which the above equivalence holds.

We know that for $i = n$, we have $M \models_{g_i} ?\neg\phi$. This automatically means that for $i > n$, the above equivalence will not hold either, since we cannot satisfy $?\phi$. Thus, we have exactly one n. □

[1] In fact, as we already mentioned in Chapter 2, we can ask ourselves if it is reasonable to talk about side effects in non-deterministic programs. We have left this question for future work.

[2] That is unless we are dealing with an infinite loop, but in that case the program has no evaluation and we are not interested in those.

The second interesting property of a deterministic program is the following:

Definition 7. *A deterministic program $d\pi$ is said to be in **canonical form** if only concatenations occur as composition operators.*

This property is going to be very useful, because we can prove that given an initial valuation g, any program has a unique canonical form that has the same behavior:

Proposition 4. *Let $d\pi$ be a deterministic program. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$. There is a unique deterministic program $d\pi'$ in canonical form such that*

$$ {}_g[\![d\pi]\!]^M_h \text{ iff } {}_g[\![d\pi']\!]^M_h $$

and $d\pi'$ executes the same basic instructions and the same number of basic instructions as $d\pi$.

Proof. If $d\pi = (?\phi; d\pi_1) \cup (?\neg\phi; d\pi_2)$, then $d\pi'$ depends on the truth of ϕ:

$$ d\pi' = \begin{cases} ?\phi; d\pi'_1 & \text{if } M \models_g \phi \\ ?\neg\phi; d\pi'_2 & \text{o.w.} \end{cases} $$

By induction we can assume that $d\pi'_1$ and $d\pi'_2$ are the canonical forms of $d\pi_1$ and $d\pi_2$ (if these are not empty), respectively. The truth of ${}_g[\![d\pi]\!]^M_h$ iff ${}_g[\![d\pi']\!]^M_h$ follows directly from QDL13 in this case.

If $d\pi = (?\phi; d\pi_1)^*; ?\neg\phi$, we need to use n as meant in Proposition 3:

$$ d\pi' = (?\phi; d\pi'_1)^n; ?\neg\phi $$

Once again we can assume by induction that $d\pi'_1$ is the canonical form of $d\pi_1$ (once again if $d\pi_1$ is not empty). The truth of ${}_g[\![d\pi]\!]^M_h$ iff ${}_g[\![d\pi']\!]^M_h$ now follows directly from Proposition 3.

It is easy to see that in both these cases, $d\pi'$ executes the same basic instructions as $d\pi$. It is also easy to see that $d\pi'$ is unique: we cannot add instructions using union or repetition because then $d\pi'$ will no longer be in canonical form and we cannot add instructions using concatenation because those instructions will be executed, which violates the requirement that $d\pi'$ only executes the same basic instructions as $d\pi$. We cannot alter or remove instructions in $d\pi'$ either because all instructions in $d\pi'$ get executed, so altering or removing one would also violate said requirement. □

5 The logic of formulas in $\mathsf{DLA_f}$

Now that we have $\mathrm{DLA_f}$ defined and shown how it works, it is time to examine the logic of formulas a little closer. As we have mentioned before, we are making use of short-circuit versions of the $\wedge$ and $\vee$ connectives, i.e. connectives that prescribe short-circuit evaluation. In [5], different flavours of short-circuit logics (logics that can be defined by short-circuit evaluation) are identified. In this chapter we will give a short overview of these and present the short-circuit logic that underlies the formulas in $\mathrm{DLA_f}$, which turns out to be repetition-proof short-circuit logic (RPSCL).

5.1 Proposition algebra

Short-circuit logic can be defined using *proposition algebra*, an algebra that has short-circuit evaluation as its natural semantics. Proposition algebra is introduced by Bergstra and Ponse in [4] and makes use of Hoare's ternary connective $x \lhd y \rhd z$, which is called the *conditional* [16]. A more common expression for this conditional is *if* y *then* x *else* z, with x, y and z ranging of propositional statements (including propositional variables). Throughout this thesis, we will use *atom* as a shorthand for propositional variable.

Using a signature which includes this conditional, $\Sigma_{\mathrm{CP}} = \{\top, \bot, \_ \lhd \_ \rhd \_\}$, the following set CP of axioms for proposition algebra can be defined:

$$x \lhd \top \rhd y = x \tag{CP1}$$
$$x \lhd \bot \rhd y = y \tag{CP2}$$
$$\top \lhd x \rhd \bot = x \tag{CP3}$$
$$x \lhd (y \lhd z \rhd u) \rhd v = (x \lhd y \rhd v) \lhd z \rhd (x \lhd u \rhd v) \tag{CP4}$$

In the earlier mentioned paper [4], varieties of so-called *valuation algebras* are defined that serve the interpretation of a logic over Σ_{CP} by means of short-circuit evaluation. The evaluation of the conditional $t_1 \lhd t_2 \rhd t_3$ is then as follows: first t_2 gets evaluated. That yields either T, in which case the final evaluation result is determined by the evaluation of t_1, or F, in which case the same goes for t_3.

All varieties mentioned in [4] satisfy the above four axioms. The most distinguishing variety is called the variety of *free reactive valuations* and is ax-

iomatized by exactly the four axioms above (further referred to as *conditional propositions* (CP)) and nothing more. The associated valuation congruence is called free valuation congruence and written as $=_{fr}$. Thus, for each pair of closed terms[1] t, t' over Σ_{CP}, we have

$$\mathrm{CP} \vdash t = t' \iff t =_{fr} t'$$

Using the conditional, we can define negation ($\neg$), left-sequential conjunction ($\mathbin{\owedge}$) and left-sequential disjunction ($\mathbin{\ovee}$) as follows:

$$\begin{aligned} \neg x &= \bot \lhd x \rhd \top \\ x \mathbin{\owedge} y &= y \lhd x \rhd \bot \\ x \mathbin{\ovee} y &= \top \lhd x \rhd y \end{aligned}$$

The above defined connectives are associative and each other's dual. In CP, it is not possible to express the conditional $x \lhd y \rhd z$ using any set of Boolean connectives, such as $\mathbin{\owedge}$ and $\mathbin{\ovee}$ [4].

By adding axioms to CP, it can be strengthened. The signature and axioms of one such extension are called *memorizing CP*. We write CP_{mem} for this extension that is obtained by adding the axiom CPmem to CP. This axiom expresses that the first evaluation value of y is memorized:

$$x \lhd y \rhd (z \lhd u \rhd (v \lhd y \rhd w)) = x \lhd y \rhd (z \lhd u \rhd w) \tag{CPmem}$$

With $u = \bot$ and by replacing y by $\neg y$ we get the contraction law:

$$(w \lhd y \rhd v) \lhd y \rhd x = w \lhd y \rhd x$$

A consequence of contraction is the idempotence of $\mathbin{\owedge}$. Furthermore, CP_{mem} is the least identifying extension of CP for which the conditional can be expressed using negation, conjunction and disjunction. To be exact, the following holds in CP_{mem}:

$$x \lhd y \rhd z = (y \mathbin{\owedge} x) \mathbin{\ovee} (\neg y \mathbin{\owedge} z)$$

We write $=_{mem}$ (memorizing valuation congruence) for the valuation congruence axiomatized by CP_{mem}.

Another extension of CP, the most identifying one distinguised in [4], is defined by adding both the contraction law and the axiom below, which expresses how the order of u and y can be swapped, to CP:

$$(x \lhd y \rhd z) \lhd u \rhd v = (x \lhd u \rhd v) \lhd y \rhd (z \lhd u \rhd v) \tag{CPstat}$$

The signature and axioms of this extension, for which we write CP_{stat}, are called *static CP*. We write $=_{stat}$ (*static valuation congruence*) for the valuation congruence axiomatized by CP_{stat}. A consequence in CP_{stat} is $v = v \lhd y \rhd v$, which can be used to derive the commutativity of $\mathbin{\owedge}$: $x \mathbin{\owedge} y = y \mathbin{\owedge} x$.

CP_{stat} is the most identifying extension of CP because it is 'equivalent with' propositional logic, that is, all tautologies in propositional logic can be proved in CP_{stat} using the above translations of its common connectives [5].

[1] Terms that may contain atoms, but not variables.

5.2 Short-Circuit Logics

In this section we will present the definition of short-circuit logic and its most basic form, free short-circuit logic (FSCL). The definitions are given using *module algebra* [2]. In module algebra, $S \,\square\, X$ is the operation that exports the signature S from module X while declaring other signature elements hidden. Using this operation, short-circuit logics are defined as follows:

Definition 8. *A **short-circuit logic** is a logic that implies the consequences of the module expression*

$$\begin{aligned}\mathrm{SCL} = \{\top, \neg, \mathbin{\diamondsuit\!\!\wedge}\} \,\square\, (\mathrm{CP} \\ + (\neg x = \bot \lhd x \rhd \top) \\ + (x \mathbin{\diamondsuit\!\!\wedge} y = y \lhd x \rhd \bot))\end{aligned}$$

Thus, the conditional composition is declared to be an auxiliary operator. In SCL, $\bot$ can be used as a shorthand for $\neg\top$. After all, we have that

$$\mathrm{CP} + (\neg x = \bot \lhd x \rhd \top) \vdash \bot = \neg\top$$

With this definition, we can immediately define the most basic short-circuit logic we distinguish:

Definition 9. FSCL ***(free short-circuit logic)*** *is the short-circuit logic that implies no other consequences than those of the module expression* SCL*.*

Using these definitions we can provide equations that are derivable from FSCL. The question whether a finite axiomatization of FSCL with only sequential conjunction, negation and $\top$ exists, is open, but the following set EqFSCL of equations for FSCL is proposed in [5]:[2]

$$\bot = \neg\top \tag{SCL1}$$
$$x \mathbin{\diamondsuit\!\!\vee} y = \neg(\neg x \mathbin{\diamondsuit\!\!\wedge} \neg y) \tag{SCL2}$$
$$\neg\neg x = x \tag{SCL3}$$
$$\top \mathbin{\diamondsuit\!\!\wedge} x = x \tag{SCL4}$$
$$x \mathbin{\diamondsuit\!\!\wedge} \top = x \tag{SCL5}$$
$$\bot \mathbin{\diamondsuit\!\!\wedge} x = \bot \tag{SCL6}$$
$$(x \mathbin{\diamondsuit\!\!\wedge} y) \mathbin{\diamondsuit\!\!\wedge} z = x \mathbin{\diamondsuit\!\!\wedge} (y \mathbin{\diamondsuit\!\!\wedge} z) \tag{SCL7}$$
$$(x \mathbin{\diamondsuit\!\!\vee} y) \mathbin{\diamondsuit\!\!\wedge} (z \mathbin{\diamondsuit\!\!\wedge} \bot) = (\neg x \mathbin{\diamondsuit\!\!\vee} (z \mathbin{\diamondsuit\!\!\wedge} \bot)) \mathbin{\diamondsuit\!\!\wedge} (y \mathbin{\diamondsuit\!\!\wedge} (z \mathbin{\diamondsuit\!\!\wedge} \bot)) \tag{SCL8}$$
$$(x \mathbin{\diamondsuit\!\!\vee} y) \mathbin{\diamondsuit\!\!\wedge} (z \mathbin{\diamondsuit\!\!\vee} \top) = (x \mathbin{\diamondsuit\!\!\wedge} (z \mathbin{\diamondsuit\!\!\vee} \top)) \mathbin{\diamondsuit\!\!\vee} (y \mathbin{\diamondsuit\!\!\wedge} (z \mathbin{\diamondsuit\!\!\vee} \top)) \tag{SCL9}$$
$$((x \mathbin{\diamondsuit\!\!\wedge} \bot) \mathbin{\diamondsuit\!\!\vee} y) \mathbin{\diamondsuit\!\!\wedge} z = (x \mathbin{\diamondsuit\!\!\wedge} \bot) \mathbin{\diamondsuit\!\!\vee} (y \mathbin{\diamondsuit\!\!\wedge} z) \tag{SCL10}$$

Note that equations SCL2 and SCL3 imply a left-sequential version of De Morgan's laws.

An important equation that is absent is the following:

$$x \mathbin{\diamondsuit\!\!\wedge} \bot = \bot$$

[2] In [5] it is stated that the authors did not find any equations derivable from FSCL but not from EqFSCL.

This is what we would expect, since evaluation of $t \mathbin{\curlywedge} \bot$ (with t a closed term) can generate a side effect that is absent in the evaluation of $\bot$, although we know that evaluation of $t \mathbin{\curlywedge} \bot$ always yields F.

We now have the most basic short-circuit logic and some of its equations defined, but of course there also is a "most liberal" short-circuit logic below propositional logic. This logic is based on memorizing CP and satisfies idempotence of $\curlywedge$ (and $\curlyvee$), but not its commutativity. It is defined as follows:

Definition 10. MSCL ***(memorizing short-circuit logic)*** *is the short-circuit logic that implies no other consequences than those of the module expression*

$$\begin{aligned}\{\top, \neg, \curlywedge\} \,\square\, (&\mathrm{CP}_{mem} \\ &+ (\neg x = \bot \lhd x \rhd \top) \\ &+ (x \mathbin{\curlywedge} y = y \lhd x \rhd \bot))\end{aligned}$$

For the set of axioms EqMSCL, intuitions and an example, and a completeness proof of MSCL we refer the reader to [5]. Adding the axiom $x \mathbin{\curlywedge} \bot = \bot$ to MSCL, or equivalently, the axiom $\bot \lhd x \rhd \bot = \bot$ to CP_{mem}, yields so-called static short-circuit logic (SSCL), which is equivalent with propositional logic (be it in sequential notation and defined by short-circuit evaluation).

Definition 11. SSCL ***(static short-circuit logic)*** *is the short-circuit logic that implies no other consequences than those of the module expression*

$$\begin{aligned}\{\top, \neg, \curlywedge\} \,\square\, (&\mathrm{CP}_{mem} \\ &+ (\bot \lhd x \rhd \bot = \bot) \\ &+ (\neg x = \bot \lhd x \rhd \top) \\ &+ (x \mathbin{\curlywedge} y = y \lhd x \rhd \bot))\end{aligned}$$

5.3 Repetition-Proof Short-Circuit Logic

With both the most basic as well as the most liberal short-circuit logic we distinguish defined, we can present the variant of short-circuit logic that we are interested in because it underlies the logic of formulas in $\mathrm{DLA_f}$: *repetition-proof short-circuit logic* (RPSCL). This SCL-variant stems from an axiomatization of proposition algebra called repetition-proof CP (CP_{rp}) that is in between CP and CP_{mem} and involves explicit reference to a set A of atoms (propositional variables).

The axiom system CP_{rp} is defined as the extension of CP with the following two axiom schemes (for $a \in A$), which imply that any subsequent evaluation result of an atom a equals the current one:

$$(x \lhd a \rhd y) \lhd a \rhd z = (x \lhd a \rhd x) \lhd a \rhd z \qquad \text{(CPrp1)}$$

$$x \lhd a \rhd (y \lhd a \rhd z) = x \lhd a \rhd (z \lhd a \rhd z) \qquad \text{(CPrp2)}$$

We write $\mathrm{Eq}_{rp}(A)$ to denote the set of these axioms schemes in the format of module algebra. In CP_{rp} the conditional cannot be expressed in terms of $\curlywedge$, $\neg$ and $\top$: in [4] it is shown that the propositional statement $a \lhd b \rhd c$ (for atoms $a, b, c \in A$) cannot be expressed modulo repetition-proof valuation congruence, that is, the valuation congruence axiomatized by CP_{rp}. The definition of RPSCL then becomes:

Definition 12. RPSCL ***(repetition-proof short-circuit logic)*** *is the short-circuit logic that implies no other consequences than those of the module expression*

$$\{\top, \neg, \curlywedge, a \mid a \in A\} \ \square \ (\mathrm{CP} + \mathrm{Eq}_{rp}(A) + (\neg x = \bot \lhd x \rhd \top) + (x \curlywedge y = y \lhd x \rhd \bot))$$

The equations defined by RPSCL include those that are defined by EqFSCL as well as for $a \in A$:

$$a \curlywedge (a \curlyvee x) = a \curlywedge (a \curlyvee y) \tag{RP1}$$
$$a \curlyvee (a \curlywedge x) = a \curlyvee (a \curlywedge y) \tag{RP2}$$
$$(a \curlyvee \neg a) \curlywedge x = (\neg a \curlywedge a) \curlyvee x \tag{RP3}$$
$$(\neg a \curlyvee a) \curlywedge x = (a \curlywedge \neg a) \curlyvee x \tag{RP4}$$
$$(a \curlywedge \neg a) \curlywedge x = a \curlywedge \neg a \tag{RP5}$$
$$(\neg a \curlywedge a) \curlywedge x = \neg a \curlywedge a \tag{RP6}$$
$$(x \curlyvee y) \curlywedge (a \curlywedge \neg a) = (\neg x \curlyvee (a \curlywedge \neg a)) \curlywedge (y \curlywedge (a \curlywedge \neg a)) \tag{RP7}$$
$$(x \curlyvee y) \curlywedge (\neg a \curlywedge a) = (\neg x \curlyvee (\neg a \curlywedge a)) \curlywedge (y \curlywedge (\neg a \curlywedge a)) \tag{RP8}$$
$$(x \curlyvee y) \curlywedge (a \curlyvee \neg a) = (x \curlywedge (a \curlyvee \neg a)) \curlyvee (y \curlywedge (a \curlyvee \neg a)) \tag{RP9}$$
$$(x \curlyvee y) \curlywedge (\neg a \curlyvee a) = (x \curlywedge (\neg a \curlyvee a)) \curlyvee (y \curlywedge (\neg a \curlyvee a)) \tag{RP10}$$
$$((a \curlywedge \neg a) \curlyvee y) \curlywedge z = (a \curlywedge \neg a) \curlyvee (y \curlywedge z) \tag{RP11}$$
$$((\neg a \curlywedge a) \curlyvee y) \curlywedge z = (\neg a \curlywedge a) \curlyvee (y \curlywedge z) \tag{RP12}$$

It is an open question whether the equations SCL1-SCL10 and the equation schemes RP1-RP12 axiomatize RPSCL, but it will be shown below that RPSCL is the logic that models equivalence of formulas in $\mathrm{DLA_f}$, where

$$A = \{Rt_1 \dots t_n, t_1 = t_2, [v := t]\top\}$$

For this reason, we add the conditional $\phi_1 \lhd \phi_2 \rhd \phi_3$ and the constant $\bot$ to $\mathrm{DLA_f}$ (thus making $\curlyvee$ and $\neg$ definable). In order to decide whether different $\mathrm{DLA_f}$ formulas are equivalent, just translate these to CP_{rp} and decide their equivalence (either by axiomatic reasoning or by checking their repetition-proof valuation congruence). So, we extend the formulas in $\mathrm{DLA_f}$ in order to characterize the logic that models their equivalence. In this extension of $\mathrm{DLA_f}$, which we baptize $\mathrm{DLCA_f}$ (for Dynamic Logic with the Conditional and Assignments as Formulas), truth in M relative an initial valuation g for the conditional is defined as follows:

$$M \models_g (\phi_2 \lhd \phi_1 \rhd \phi_3) \text{ iff for } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M, \begin{cases} M \models_h \phi_2 & \text{if } M \models_g \phi_1 \\ M \models_h \phi_3 & \text{o.w.} \end{cases} \tag{DLCA}$$

This means that we need an extra equation for the program extraction function Π too which handles the conditional. For model M, initial valuation g and ${}_g[\![\phi_1]\!]_h^M$

$$\Pi_g^M(\phi_2 \lhd \phi_1 \rhd \phi_3) = \begin{cases} \Pi_g^M(\phi_1); \Pi_h^M(\phi_2) & \text{if } M \models_g \phi_1 \\ \Pi_g^M(\phi_1); \Pi_h^M(\phi_3) & \text{o.w.} \end{cases}$$

In the remainder of this section we consider formulas over this signature, thus formulas over A composed with $\_ \lhd \_ \rhd \_$. Below we will prove for all mentioned axioms that they are valid in $\mathrm{DLCA_f}$.

Proposition 5. *Let M be a model for $DLCA_f$. The axiom CP1, that is*

$$x \lhd \top \rhd y = x \tag{CP1}$$

is valid in M.

Proof. Let t_1, t_2 be arbitrary formulas and let g be an initial valuation. Regardless of g, we have $M \models_g \top$ (by QDL3), so by DLCA, we get $M \models_g (t_1 \lhd \top \rhd t_2)$ iff for ${}_g[\![?\top]\!]^M_h$, $M \models_h t_1$. Since $g = h$, we indeed have that $M \models_g (t_1 \lhd \top \rhd t_2)$ iff $M \models_g t_1$. □

Proposition 6. *Let M be a model for $DLCA_f$. The axiom CP2, that is*

$$x \lhd \bot \rhd y = y \tag{CP2}$$

is valid in M.

Proof. Let t_1, t_2 be arbitrary formulas and let g be an initial valuation. $\bot$ is a shorthand for $\neg\top$, so we first need QDL6, which states that $M \models_g \neg\top$ iff not $M \models_g \top$, which is never the case. So for any initial valuation g, $M \models_g \bot$ is false. Thus by DLCA, we get $M \models_g (t_1 \lhd \bot \rhd t_2)$ iff for ${}_g[\![?\top]\!]^M_h$, $M \models_h t_1$. Since $g = h$, we indeed have that $M \models_g (t_1 \lhd \bot \rhd t_2)$ iff $M \models_g t_2$. □

Proposition 7. *Let M be a model for $DLCA_f$. The axiom CP3, that is*

$$\top \lhd x \rhd \bot = x \tag{CP3}$$

is valid in M.

Proof. Let t be an arbitrary formula and let g be an initial valuation. If $M \models_g t$ then by DLCA we get for ${}_g[\![\Pi^M_g(t)]\!]^M_h$, $M \models_h \top$, which also is true. If $M \not\models_g t$ then by DLCA we obtain $M \models_h \bot$ (note that also in this case, h is defined), which also is false. Thus $M \models_g t$ iff $M \models_g \top \lhd t \rhd \bot$ and hence the axiom CP3 is valid. □

Proposition 8. *Let M be a model for $DLCA_f$. The axiom CP4, that is*

$$x \lhd (y \lhd z \rhd v) \rhd u = (x \lhd y \rhd u) \lhd z \rhd (x \lhd v \rhd u) \tag{CP4}$$

is valid in M.

Proof. Let t_1, t_2, t_3, t_4, t_5 be arbitrary formulas and let g be an initial valuation. We are going to have to show that

$$M \models_g t_1 \lhd (t_2 \lhd t_3 \rhd t_4) \rhd t_5 \text{ iff } M \models_g (t_1 \lhd t_2 \rhd t_5) \lhd t_3 \rhd (t_1 \lhd t_4 \rhd t_5)$$

We have to apply DLCA multiple times here. By applying it to the left hand side we get for ${}_g[\![\Pi^M_g(t_2 \lhd t_3 \rhd t_4)]\!]^M_f$

$$M \models_g t_1 \lhd (t_2 \lhd t_3 \rhd t_4) \rhd t_5 \text{ iff } \begin{cases} M \models_f t_1 & \text{if } M \models_g (t_2 \lhd t_3 \rhd t_4) \\ M \models_f t_5 & \text{o.w.} \end{cases}$$

By applying DLCA again to $M \models_g (t_2 \lhd t_3 \rhd t_4)$ we get for ${}_g[\![\Pi_g^M(t_3)]\!]_{f'}^M$

$$M \models_g (t_2 \lhd t_3 \rhd t_4) \text{ iff } \begin{cases} M \models_{f'} t_2 & \text{if } M \models_g t_3 \\ M \models_{f'} t_4 & \text{o.w.} \end{cases}$$

So if $M \models_g t_3$ and $M \models_{f'} t_2$, we get $M \models_f t_1$. If on the other hand $M \not\models_g t_3$ but $M \models_{f'} t_4$, we also get $M \models_f t_1$. In all other situations we get $M \models_f t_5$.

Let us now consider the right hand side of the equation. Here we get for ${}_g[\![\Pi_g^M(t_3)]\!]_{h'}^M$:

$$M \models_g (t_1 \lhd t_2 \rhd t_5) \lhd t_3 \rhd (t_1 \lhd t_4 \rhd t_5) \text{ iff } \begin{cases} M \models_{h'} (t_1 \lhd t_2 \rhd t_5) \text{ if } M \models_g t_3 \\ M \models_{h'} (t_1 \lhd t_4 \rhd t_5) \text{ o.w.} \end{cases}$$

Let us first turn our attention to the situation where $M \models_g t_3$. We need to apply DLCA again and get for ${}_{h'}[\![\Pi_{h'}^M(t_2)]\!]_h^M$

$$M \models_{h'} (t_1 \lhd t_2 \rhd t_5) \text{ iff } \begin{cases} M \models_h t_1 & \text{if } M \models_{h'} t_2 \\ M \models_h t_5 & \text{o.w.} \end{cases}$$

In the situation where $M \not\models_g t_3$, we get for ${}_{h'}[\![\Pi_{h'}^M(t_4)]\!]_{h''}^M$

$$M \models_{h'} (t_1 \lhd t_4 \rhd t_5) \text{ iff } \begin{cases} M \models_{h''} t_1 & \text{if } M \models_{h'} t_4 \\ M \models_{h''} t_5 & \text{o.w.} \end{cases}$$

So on the right hand side, if $M \models_g t_3$ and $M \models_{h'} t_2$, we get $M \models_h t_1$. If $M \not\models_g t_3$ but $M \models_{h'} t_4$, we also get $M \models_{h''} t_1$. In the other situations we get either $M \models_h t_5$ or $M \models_{h''} t_5$.

To prove that is the same result as on the left-hand side, we need to prove that $f' = h'$, $f = h$ if $M \not\models_g t_3$, and $f = h''$ if $M \models_g t_3$. The last two statements seem contradictory, but as we will see f can actually take two different valuations depending on the truth of t_3. The mentioned variations are all determined using the program extraction function. To recap, we have the following:

$$\begin{gathered} {}_g[\![\Pi_g^M(t_2 \lhd t_3 \rhd t_4)]\!]_f^M \\ {}_g[\![\Pi_g^M(t_3)]\!]_{f'}^M \\ {}_g[\![\Pi_g^M(t_3)]\!]_{h'}^M \\ {}_{h'}[\![\Pi_{h'}^M(t_2)]\!]_h^M \\ {}_{h'}[\![\Pi_{h'}^M(t_4)]\!]_{h''}^M \end{gathered}$$

We can immediately see that $f' = h'$. Using the updated definition for the program extraction function we get that

$${}_g[\![\Pi_g^M(t_2 \lhd t_3 \rhd t_4)]\!]_f^M \text{ iff } \begin{cases} {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_2)]\!]_f^M & \text{if } M \models_g t_3 \\ {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_4)]\!]_f^M & \text{o.w.} \end{cases}$$

Using the new rule for the conditional, we get that:

$$\begin{gathered} {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_2)]\!]_f^M \text{ if } M \models_g t_3 \\ {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_4)]\!]_f^M \text{ if } M \not\models_g t_3 \end{gathered}$$

To determine if $f = h$, we need to have $M \models_g t_3$ and we need to evaluate:

$$ {}_g[\![\Pi_g^M(t_3)]\!]_{h'}^M \text{ and } {}_{h'}[\![\Pi_{h'}^M(t_2)]\!]_h^M $$

By QDL12, we know that is equivalent to

$$ {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_2)]\!]_h^M $$

So indeed we have that if $M \models_g t_3$, then $f = h$. Using the same argument, we get that if $M \not\models_g t_3$, then

$$ {}_g[\![\Pi_g^M(t_3); \Pi_{h'}^M(t_4)]\!]_{h''}^M $$

Therefore, if $M \not\models_g t_3$ then $f = h''$. □

With those four axioms proven, we already know for a fact that the logic of formulas in $\mathrm{DLA_f}$ indeed is a short-circuit logic. To prove that it is a repetition-free short-circuit logic, we need to prove the axiom schemes CPrp1 and CPrp2, too. Those axiom schemes make use of atoms $a \in A$.

Proposition 9. *Let M be a model for $DLCA_f$. The axiom CPrp1, that is*

$$ (x \lhd a \rhd y) \lhd a \rhd z = (x \lhd a \rhd x) \lhd a \rhd z \tag{CPrp1} $$

is valid in M.

Proof. Let t_1, t_2, t_3 be arbitrary formulas and g an initial valuation. $M \models_g a$ can either be true or false. If it is false, both the left hand side and the right hand side, by DLCA, are determined for ${}_g[\![\Pi_g^M(a)]\!]_h^M$ by $M \models_h t_3$. If it is true, the question if $M \models_h a$ is asked. We have to prove that for every atom $a \in A$, the reply to this will be the same as the reply to $M \models_g a$ (namely, true), that is:

$$ M \models_h a \text{ iff } M \models_g a $$

Recall that a can be of the forms $\{Rt'_1 \dots t'_n, t'_1 = t'_2, [v := t']\top\}$. For the first two atoms we can immediately see our claim is true, since $\Pi_g^M(a) = ?\top$ and therefore $g = h$. For $[v := t']\top$ the claim immediately follows from DLA9: it is, regardless of the valuation, always true. □

Proposition 10. *Let M be a model for $DLCA_f$. The axiom CPrp2, that is*

$$ x \lhd a \rhd (y \lhd a \rhd z) = x \lhd a \rhd (z \lhd a \rhd z) \tag{CPrp2} $$

is valid in M.

Proof. This is the symmetric variant of CPrp1 and proven similarly. □

By proving the validity of these axiom schemes in $\mathrm{DLCA_f}$ we have proven that the equations SCL1-SCL10 together with RP1-RP12 are axioms for formulas in $\mathrm{DLCA_f}$. CP_{rp} indeed is the most identifying extension of CP which is valid for formulas. After all, the first more identifying extension we distinguish is CP_{con} (*contractive CP*) [5], from which amongst others the following weak contraction rule can be derived: for $a \in A$

$$ a \curlywedge a = a $$

Clearly this is not valid for $\mathrm{DLA_f}$-formulas such as $[x := x + 1]\top$.

6 A treatment of side effects

6.1 Introduction

Now that we have defined a system to model program instructions and program states, we can return to our original problem: that of formally defining side effects. Like I said in Section 2.1, the basic idea is that a side effect has occurred in the execution of a program if there is a difference between the actual evaluation and the expected evaluation of a program given an initial valuation.

We can immediately see however, that we cannot build a definition of side effects based on the actual and expected evaluation of an entire program. Such a definition will get into trouble when there are multiple side effects, especially if those cancel each other out or reinforce each other. Consider for example the following program:

$$\pi = ?([x := x + 1]\top); ?([x := x + 1]\top)$$

If we are only going to look at the entire program, we will detect one side effect here, that has incremented the value of x by two. However, it appears to be more acceptable to say that *two* side effects have occurred, that happen to affect the same value.

It gets even more interesting if there is a formula in between the two clauses above and the clauses themself cancel each other out:

$$\pi = ?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} \phi \mathbin{\diamond\!\!\wedge} [x := x \dot{-} 1]\top)$$

If we again only look at the entire program, we will detect no side effects (unless side effects occur in ϕ). However, because ϕ might use or modify x as well, it seems we will have to pay attention to the side effect of the first clause, even though it will be cancelled out on by the last clause.

So instead of building a definition of side effects by looking only at the actual and expected evaluation of an entire program, we are going to build it up starting at the instruction level.

6.2 Side effects in single instructions

As said, we are going to use a bottom-up approach to define side effects, so we will first define side effects for single instructions, then move up to basic instructions and end with a full definition of side effects for programs.

The idea is that the side effect of a single instruction is the difference between the actual and expected evaluation of a single instruction. This difference is essentially the difference between the resulting valuations after, respectively, the actual and expected evaluation of the single instruction. The difference between two valuations is defined as follows:

Definition 13. *Given a model M, the **difference** between valuations g and h is defined as those variables that have a different assignment in g and h:*

$$(x \mapsto k') \in \delta^M(g,h) \text{ iff } g(x) = k, h(x) = k' \text{ and } M \not\models k = k'$$

This notion of difference is not symmetric.

We already know what the actual evaluation of a single instruction is: for this we can use $\mathrm{DLA_f}$. This leaves us to define the expected evaluation. For this we need to know for each single instruction how we expect it to evaluate, that is, what changes we expect it to make to the initial valuation. We have the following expectations of each single instruction:

- **Assignments** change the initial valuation by updating the variable assignment of the variable under consideration to the (interpretation of the) new variable assignment.
- **Tests** do not change the initial valuation: they only yield T or F and steer the rest of the program accordingly.

We need the following equations for determining the expected evaluation $\mathcal{E}$ of a single instruction:

$$M \models^{\mathcal{E}}_g \top \text{ always} \tag{EV1}$$

$$M \models^{\mathcal{E}}_g Rt_1 \dots t_n \text{ iff } (\llbracket t_1 \rrbracket^M_g, \dots, \llbracket t_n \rrbracket^M_g) \in R^M \tag{EV2}$$

$$M \models^{\mathcal{E}}_g t_1 = t_2 \text{ iff } \llbracket t_1 \rrbracket^M_g \text{ is the same as } \llbracket t_2 \rrbracket^M_g \tag{EV3}$$

$$M \models^{\mathcal{E}}_g [v := t]\top \text{ always} \tag{EV4}$$

$${}_g\llbracket v := t \rrbracket^{M,\mathcal{E}}_h \text{ iff } h = g[v \mapsto \llbracket t \rrbracket^M_g] \tag{EV5}$$

$${}_g\llbracket ?\varphi \rrbracket^{M,\mathcal{E}}_h \text{ iff } g = h \text{ and } M \models^{\mathcal{E}}_g \varphi \tag{EV6}$$

Now that we have the actual and the expected evaluation of a single instruction, we can define its side effects. As said, this is going to be the difference between the two resulting valuations.

Definition 14. *Let ρ be a single instruction. Let model M be given and let g be an initial valuation. Furthermore, let h be a valuation such that ${}_g\llbracket \rho \rrbracket_h$ and let h' be a valuation such that ${}_g\llbracket \rho \rrbracket^{\mathcal{E}}_{h'}$. The set of side effects of single instruction ρ given model M and initial valuation g is defined as*

$$\mathcal{S}^M_g(\rho) = \delta^M(h', h)$$

It is important to note that the valuations h and h' as meant in the above definition may not exist. We are not interested in those situations, however. If h and h' do exist, they are unique. Also note that $\delta^M(h', h)$ returns the variable assignment of valuation h if there is a difference with the variable assignment of valuation h'. Thus, the set of side effects is defined as a set containing those variables that have a different assignment after the actual and expected valuation, with as assignments the ones the variables actually get (that is, the assignments they will have after evaluating the single instruction with the actual evaluation).

We will illustrate this with two examples. First, consider the single instruction $\rho = (x := 1)$, evaluated under model M in initial valuation g with $g(x) = 0$. We want to know if this causes a side effect, so we need to know the actual evaluation and the expected evaluation. To calculate the actual evaluation, we need to know if ${}_g[\![x := 1]\!]^M_h$ and if yes, for which valuation h. The equations for $\mathrm{DLA_f}$ immediately give us the answer, in this case via QDL10: $h = g[x \mapsto [\![1]\!]^M_g]$. So we get $h(x) = 1$.

Getting the expected evaluation works in a similar fashion, but instead of $\mathrm{DLA_f}$ we now use the equations above to evaluate ρ. Since the equation for evaluating an assignment (EV5) is the same as QDL10, we now get the exact same expected evaluation as the actual evaluation. Thus we get $h' = g[x \mapsto [\![1]\!]^M_g]$ and therefore $h'(x) = 1$. We can immediately see that this results in the set of side effects being empty:

$$\mathcal{S}^M_g(x := 1) = \delta^M(h', h) = \emptyset$$

This is of course what we would expect: an assignment should not have a side effect if it does not occur in a steering fragment. Let us now consider an example where we do expect a side effect: namely if an assignment does occur in a steering fragment: $\rho = ?([x := 1]\top)$. We use the same initial valuation g. First we try to find the actual evaluation again, which we do by evaluating ${}_g[\![?([x := 1]\top)]\!]^M_h$. We now need DLA11, which tells us that (in this case) ${}_g[\![?([x := 1]\top)]\!]^M_h$ iff $M \models_g ([x := 1]\top)$ and ${}_g[\![\Pi^M_g([x := 1]\top)]\!]^M_h = {}_g[\![x := 1]\!]^M_h$. Both evaluate to true, the latter with $h = g[x \mapsto 1]$.

The expected update once again takes us to the equations above; we need to determine h' such that ${}_g[\![?([x := 1]\top)]\!]^{M,\mathcal{E}}_{h'}$. For tests, the demands are fairly simple: $g = h'$ and $M \models^{\mathcal{E}}_g [x := 1]\top$ (see EV6). The latter is by EV4 defined to be always true. As a result, we get $h'(x) = g(x) = 0$. Thus we get the following set of side effects:

$$\begin{aligned}\mathcal{S}^M_g(?[x := 1]\top) &= \delta^M(h', h)\\ &= \{x \mapsto 1\}\end{aligned}$$

Again, this is exactly what we want: since we expect formulas to only yield true or false, the change this formula makes to the program state upon evaluation is a side effect.

6.3 Side effects in basic instructions

With side effects for single instructions defined, we can move up a step to side effects in basic instructions. The difference between single and basic instructions

is that in basic instructions, complex steering fragments are allowed. This means that we are going to have to define how side effects are handled in tests that contain a disjunction ($\overset{\circ}{\vee}$), conjunction ($\overset{\circ}{\wedge}$) or negation ($\neg$). The idea is that the set of side effects of the whole formula is the union of the sets of side effects of its primitive parts. However, we also have to pay attention to the short-circuit character of $\overset{\circ}{\vee}$. Only the primitive formulas that get evaluated can contribute to the set of side effects.

With this in mind, we can give the definition for side effects in (possibly) complex steering fragments. Like before, we are only interested in the side effects if the test actually succeeds. We need to define this for disjunctions, conjuctions and negations:

Definition 15. *Let $\phi = \phi_1 \overset{\circ}{\vee} \phi_2$ be a disjunction. Let model M and initial valuation g be given, with $M \models_g \phi$ and where ϕ is in its normal form. Furthermore, let f be the valuation after evaluation of formula ϕ_1, that is, ${}_g[\![?\phi_1]\!]^M_f$. The set of side effects $\mathcal{S}^M_g(?\phi)$ is defined as:*

$$\mathcal{S}^M_g(?\phi) = \begin{cases} \mathcal{S}^M_g(?\phi_1) & \text{if } M \models_g \phi_1 \\ \mathcal{S}^M_g(?\phi_1) \cup \mathcal{S}^M_f(?\phi_2) & \text{o.w.} \end{cases}$$

The case distinction is in place because of the short-circuit character of $\overset{\circ}{\vee}$. For the definition of its dual $\overset{\circ}{\wedge}$ we do not need this case distinction, because since we are again only interested in the side effects if the (entire) formula succeeds, all the formulas in the conjunction have to yield true. Therefore, the definition for conjunction is a bit easier:

Definition 16. *Let $\phi = \phi_1 \overset{\circ}{\wedge} \phi_2$ be a conjunction. Let model M and initial valuation g be given, with $M \models_g \phi$ and where ϕ is in its normal form. Furthermore, let f be the valuation after evaluation of primitive formula ϕ_1, that is, ${}_g[\![?\phi_1]\!]^M_f$. The set of side effects $\mathcal{S}^M_g(?\phi)$ is defined as:*

$$\mathcal{S}^M_g(?\phi) = \mathcal{S}^M_g(?\phi_1) \cup \mathcal{S}^M_f(?\phi_2)$$

The recursive definitions for disjunction and conjunction work because eventually, a primitive formula will be encountered, for which the side effects are already defined. Unfortunately, we cannot use a similar construction for negation. This is because the side effects in a primitive formula are only defined if that formula yields true upon evaluation, so we cannot simply treat negation as a transparent operator (that is, it is typically not true that $\mathcal{S}^M_g(\neg\phi) = \mathcal{S}^M_g(\phi)$). So we will have to define negation the hard way instead. Because we are using formulas in normal form in the other definitions, we only have to define negation for primitive formulas:

Definition 17. *Let $\neg\varphi$ be a negation. Let model M be given and let g be an initial valuation. Furthermore, let h be a valuation such that ${}_g[\![?\neg\varphi]\!]_h$ and let h' be a valuation such that ${}_g[\![?\neg\varphi]\!]^{\mathcal{E}}_{h'}$. The set of side effects of basic instruction $?\neg\varphi$ given model M and initial valuation g is defined as*

$$\mathcal{S}^M_g(?\neg\varphi) = \delta^M(h', h)$$

Now that we have a definition for side effects in (complex) steering fragments, the extension of our definition of side effects in single instructions to side effects in basic instructions is trivial:

Definition 18. *Let ϖ be a basic instruction. Let model M and initial valuation g be given and let h be a valuation such that ${}_g[\![\varpi]\!]^M_h$. The set of side effects $\mathcal{S}^M_g(\varpi)$ is defined as:*

$$\mathcal{S}^M_g(\varpi) = \begin{cases} \mathcal{S}^M_g(\rho) & \text{if } \varpi = \rho \\ \mathcal{S}^M_g(?\phi) & \text{if } \varpi = ?\phi' \text{ and } \phi \text{ is the normal form of } \phi' \end{cases}$$

We can illustrate this with a simple, yet interesting example. Consider the following basic instruction: $\varpi = ?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} [x := x \dot{-} 1]\top)$ with initial valuation g such that $g(x) = 1$. In this situation we have two side effects that happen to cancel each other out. The resulting valuation after the actual evaluation of this basic instruction will be the same as the initial valuation g.

First we observe that the formula in this basic instruction is in its normal form, a trivial observation since no negations occur in it. There are two primitive formulas in this conjunction, so the set of side effects is:

$$\begin{aligned} \mathcal{S}^M_g(?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} [x := x \dot{-} 1]\top)) = {} & \mathcal{S}^M_g(?([x := x + 1]\top)) \cup \\ & \mathcal{S}^M_{g_1}(?([x := x \dot{-} 1]\top)) \end{aligned}$$

Here g_1 is determined by ${}_g[\![?([x := x + 1]\top)]\!]^M_{g_1}$, so we get $g_1(x) = 2$. We have already seen in the previous section how the parts of the union above evaluate, so we get:

$$\begin{aligned} \mathcal{S}^M_g(?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} [x := x \dot{-} 1]\top)) &= \{x \mapsto 2\} \cup \{x \mapsto 1\} \\ &= \{x \mapsto 2, x \mapsto 1\} \end{aligned}$$

So with this definition we have avoided the trap of not detecting any side effects when there are two side effects that cancel each other out. Instead we have two side effects here, the last of which happens to restore the valuation of x to its original one.

6.4 Side effects in programs

If we are going to extend our definition to that of side effects in programs, we are going to have to define how concatenation, union and repetition are handled.

Defining side effects for entire programs is more complicated than defining side effects for single and basic instructions. This is because two composition operators, namely union and repetition, can be non-deterministic. As we have mentioned before, however, we are only interested in (the side effects of) deterministic programs. This leaves us to define how side effects are calculated for the composition operators of deterministic programs. For concatenation, this is trivial. We once again require that the entire program can be evaluated with the given initial valuation. The set of side effects of a program then is the union of the side effects in its basic instructions that are executed given some initial valuation:

Definition 19. *Let $d\pi = d\pi_1; d\pi_2$ be a deterministic program. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$.*

Furthermore, let f be the valuation such that ${}_g[\![d\pi_1]\!]^M_f$. The set of side effects $\mathcal{S}^M_g(d\pi)$ is defined by:

$$\mathcal{S}^M_g(d\pi) = \mathcal{S}^M_g(d\pi_1) \cup \mathcal{S}^M_f(d\pi_2)$$

This works in a similar fashion as the definition of side effects in complex steering fragments. We can return now to our example given in the Introduction of this chapter: $d\pi = ?([x := x + 1]\top); ?([x := x + 1]\top)$. The above definition indeed avoids the trap presented there, namely that this program only yields a single side effect. To see this, consider initial valuation g such that $g(x) = 0$. We will then get ${}_g[\![?([x := x+1]\top)]\!]^M_f$ and therefore $f(x) = 1$, so the set of side effects becomes:

$$\begin{aligned}\mathcal{S}^M_g(d\pi) &= \mathcal{S}^M_g(?([x := x + 1]\top)) \cup \mathcal{S}^M_f(?([x := x + 1]\top)) \\ &= \{x \mapsto 1\} \cup \{x \mapsto 2\} \\ &= \{x \mapsto 1, x \mapsto 2\}\end{aligned}$$

Similarly, side effects that cancel each other out, such as in $d\pi = ?([x := x + 1]\top); ?([x := x \dot{-} 1]\top)$ will now perfectly be detected, resulting for the same initial valuation g in a set of side effects $\mathcal{S}^M_g(d\pi) = \{x \mapsto 1, x \mapsto 0\}$.

Another interesting observation is that the transformation as defined in Proposition 2, which eliminates occurences of ${}_\circ\wedge$ and ${}^\circ\vee$ in steering fragments, not only preserves the relational meaning, but also the side effects of such a steering fragment. The programs $?([x := x + 1]\top \,{}_\circ\wedge\, [x := x \dot{-} 1]\top)$ and its transformed version $?([x := x+1]\top); ?([x := x \dot{-} 1]\top)$ are an illustration of this: we can easily see that both have the same set of side effects.

With concatenation defined, we can move on to the next composition operators: union and repetition. For this we can use the property that given an initial valuation, every (terminating) deterministic program has a unique canonical form that executes the same basic instructions (see Proposition 4 in Chapter 4). This makes the definition of side effects for programs containing a union or repetition straight-forward:

Definition 20. *Let $d\pi$ be a deterministic program. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$. Furthermore, let $d\pi'$ be the deterministic program in canonical form as meant in Proposition 4. The set of side effects $\mathcal{S}^M_g(d\pi)$ is defined by:*

$$\mathcal{S}^M_g(d\pi) = \mathcal{S}^M_g(d\pi')$$

We can illustrate how this works by returning to our running example, discussed in detail in Section 3.2:

$x := 1;$
IF $(x := x + 1 \,{}_\circ\wedge\, x = 2)$
THEN $y := 1$
ELSE $y := 2$

In $\mathrm{DLA_f}$, this translates to the following deterministic program $d\pi$:

$x := 1;$
$(?([x := x + 1]\top \owedge x = 2); y := 1)$
$\cup$
$(?\neg([x := x + 1]\top \owedge x = 2); y := 2)$

We have already seen that for $g(x) = g(y) = 0$, there is a valuation h such that ${}_g[\![d\pi]\!]^M_h$ (namely $h = g[x \mapsto 2, y \mapsto 1]$). We can break this program down as follows:

$$\begin{aligned} d\pi &::= \rho_1; d\pi 1 \\ \rho_1 &::= (x := 1) \\ d\pi_1 &::= (?\phi_0; \rho_2) \cup (?\neg\phi_0; \rho_3) \\ \rho_2 &::= (y := 1) \\ \rho_3 &::= (y := 2) \\ \phi_0 &::= \varphi_1 \owedge \varphi_2 \\ \varphi_1 &::= [x := x + 1]\top \\ \varphi_2 &::= (x = 2) \end{aligned}$$

We want to know the set of side effects in this program. This is determined as follows:

$$\begin{aligned} \mathcal{S}^M_g(d\pi) &= \mathcal{S}^M_g(\rho_1; d\pi 1) \\ &= \mathcal{S}^M_g(\rho_1) \cup \mathcal{S}^M_f(d\pi 1) \end{aligned}$$

where we get f by evaluating ${}_g[\![x := 1]\!]^M_f$. Thus, $f = g[x \mapsto 1]$. We can easily see that the first set of side effects $\mathcal{S}^M_g(\rho_1) = \emptyset$. The interesting part is the second set of side effects, since we now have a deterministic program of the form $d\pi_1 = (?\phi; d\pi_2) \cup (?\neg\phi; d\pi_3)$. Here $\phi = \phi_0, d\pi_2 = \rho_2$ and $d\pi_3 = \rho_3$.

We now have to ask ourselves what the canonical form of $d\pi_1$ given valuation f is. This is determined by the outcome of the test

$$?([x := x + 1]\top \owedge x = 2)$$

It is easy to see that this yields true. Thus, the canonical form $d\pi'$ of $d\pi_1$ is

$$d\pi' = ?\phi_0; \rho_2$$

Therefore according to our definition, for ${}_f[\![?\phi_0]\!]^M_h$:

$$\begin{aligned} \mathcal{S}^M_f(d\pi_1) &= \mathcal{S}^M_f(d\pi') \\ &= \mathcal{S}^M_f(?\phi_0; \rho_2) \\ &= \mathcal{S}^M_f(?\phi_0) \cup \mathcal{S}^M_h(\rho_2) \end{aligned}$$

We can once again immediately see that the second set of side effects $\mathcal{S}^M_h(\rho_2) = \emptyset$. The first set of side effects is determined in a similar fashion as in the example in the previous section. In the end, it gives us:

$$\begin{aligned} \mathcal{S}^M_f(?\phi_1) &= \mathcal{S}^M_f(?([x := x + 1]\top \owedge (x = 2))) \\ &= \mathcal{S}^M_f(?([x := x + 1]\top) \cup \mathcal{S}^M_{f'}(?(x = 2))) \end{aligned}$$

So we again get a union of two sets of side effects, where we get f' by evaluating ${}_f[\![[x := x+1]\top]\!]^M_{f'}$. Thus, $f' = f[x \mapsto 2]$. It should be clear by now that the first set of side effects contains one side effect, namely $\{x \mapsto 2\}$, whereas the latter does not contain any side effects. This gives us as final set of side effects:

$$\begin{aligned}\mathcal{S}^M_g(d\pi) &= \mathcal{S}^M_g(\rho_1) \cup ((\mathcal{S}^M_f(?([x := x+1]\top) \cup \mathcal{S}^M_{f'}(?(x = 2)))) \cup \mathcal{S}^M_h(\rho_2)) \\ &= \emptyset \cup ((\{x \mapsto 2\} \cup \emptyset) \cup \emptyset) \\ &= \{x \mapsto 2\}\end{aligned}$$

This is exactly the side effect we have come to expect from our running example.

We can now move on to an example of side effects in programs containing a repetition. Recall that repetition is defined as follows:

$${}_g[\![\pi^*]\!]^M_h \text{ iff } g = h \text{ or } {}_g[\![\pi;\pi^*]\!]^M_h \tag{QDL14}$$

So, π either gets executed not at all or at least once. The form of programs we are interested in is

$$d\pi = (?\phi;\pi)^*;?\neg\phi$$

In this case there will only ever be exactly one situation in which the program gets evaluated (see Proposition 3 in Chapter 4). Our definition of canonical forms tells us that given an initial valuation g and n as meant in Proposition 3, the canonical form $d\pi'$ of $d\pi$ is

$$d\pi' = (\pi_r)^n;?\neg\phi$$

Using this we get the following set of side effects of a deterministic program of the above form:

$$\mathcal{S}^M_g(d\pi) = \mathcal{S}^M_g((\pi_r)^n;?\neg\phi)$$

As an example of this, we can return to a slightly modified version of the example we gave in Section 3.3.2.

$x := 0;$
$y := 0;$
WHILE $(x := x+1 \mathbin{\diamondsuit\!\!\wedge} x \leq 3)$
DO $y := y+1$

In $\mathrm{DLA_f}$, this translates to the following deterministic program $d\pi$ given model M and initial valuation g such that $g(x) = g(y) = 0$:

$$d\pi = (?([x := x+1]\top \mathbin{\diamondsuit\!\!\wedge} (x \leq 3));y := y+1)^*;?\neg([x := x+1]\top \mathbin{\diamondsuit\!\!\wedge} (x \leq 3))$$

Clearly this is a deterministic program in the form we are interested in and there is a valuation h such that ${}_g[\![d\pi]\!]^M_h$. In this case we have $\pi_r = ?\phi;y := y+1$ with $\phi = [x := x+1]\top \mathbin{\diamondsuit\!\!\wedge} (x \leq 3)$. To get the canonical form $d\pi'$ of $d\pi$, we need to find the iteration n for which $?\phi$ will succeed, but for which the test will not succeed another time. This will be for $n = 3$. After all, after three iterations we will have valuation $g_3 = g[x \mapsto 3, y \mapsto 3]$. With this valuation, the test $?([x :=$

$x+1]\top \wedge (x \leq 3))$ will fail, or to put it formally: $M \not\models_{g_3} [x := x+1]\top \wedge (x \leq 3)$. This means that we will get the following set of side effects:

$$\begin{aligned}
\mathcal{S}_g^M(d\pi) &= \mathcal{S}_g^M(d\pi') \\
&= \mathcal{S}_g^M((\pi_r)^3; ?\neg\phi) \\
&= \mathcal{S}_g^M((\pi_r)^3) \cup \mathcal{S}_{g_3}^M(?\neg\phi) \\
&= \mathcal{S}_g^M(\pi_r; \pi_r; \pi_r) \cup \mathcal{S}_{g_3}^M(?\neg\phi) \\
&= \{x \mapsto 1, x \mapsto 2, x \mapsto 3\} \cup \{x \mapsto 4\} \\
&= \{x \mapsto 1, x \mapsto 2, x \mapsto 3, x \mapsto 4\}
\end{aligned}$$

Is this the result we would expect? The answer is yes. It is clear that for each time the test is evaluated, a side effect occurs. The test is performed four times: three times it succeeds (after which the program executes the body of its loop) and the fourth time it fails, but not after updating the valuation of x. The program evaluates with as final valuation $h = g[x \mapsto 4, y \mapsto 3]$.

6.5 Side effects outside steering fragments

The keen observer will have noticed by now that under our current definition, side effects can only occur in steering fragments. I have been going through quite some trouble, however, to make my definitions of side effects as general as possible. Even though in this thesis I am only interested in side effects in steering fragments, I am fully aware that views can differ on what the main effect and what the side effect of an instruction is. That may either be a matter of opinion or a matter of necessity, as in different systems, the same instruction may have a side effect in one system and not in the other.

The way my definitions of side effects[1] are built up, one need only change the *expected evaluation* of an instruction in order to change if it is viewed as a side effect in a certain context. Consider, for example, the sometimes accepted view that an assignment causes a side effect, no matter where it occurs in a program. This view is for example expressed by Norrish in [17]. The only change we would need to make to our system to incorporate that view is a change to the expected evaluation of the assignment, which would then become:

$$_g[\![v := t]\!]_h^{M,\mathcal{E}} \text{ iff } g = h$$

The consequence of this in our current setting would be that the expected evaluation of every program always has a resulting valuation h that is equal to the initial valuation g, since only assignments can make changes to a valuation currently and by the above definition we do not expect any assignment to do so, wherever it occurs in the program. As a consequence, any change to the valuation (caused by the actual evaluation) will automatically be a side effect.

It is almost as simple to add new instructions to our setting. I definitely do not want to claim that the instructions I have defined in $\mathrm{DLA_f}$ are exhaustive, so this need may arise. If we were, for instance, to re-introduce the random assignment $v := ?$, all we would have to do was to define the actual and expected

[1] As well as the definitions of classes of side effects presented in Chapter 7.

evaluation of this. The actual evaluation is already given by Harel in [14] and Van Eijck in [11]:

$$ {}_g[\![v := ?]\!]^M_h \text{ iff } g \sim_v h $$

If we also would want to allow random assignments in tests, we would have to add a rule for that as well, similar to the one already in place for normal assignments:

$$ M \models_g [v := ?]\top \text{ iff } {}_g[\![v := ?]\!]^M_h $$

The definition of the expected evaluation is dictated by what we really expect the random assignment to do. This can be the same as what it actually does, in which case we have to define the expected evaluation to be the same as the actual evaluation above:

$$ {}_g[\![v := ?]\!]^{M,\mathcal{E}}_h \text{ iff } {}_g[\![v := ?]\!]^M_h $$
$$ M \models^{\mathcal{E}}_g [v := ?]\top \text{ iff } M \models_g [v := ?]\top $$

If we expect random assignments to do something different, all we have to do is define the expected evaluation accordingly. This expected evaluation can literally be anything: from simply not updating the valuation at all to always setting a completely unrelated variable to 42:

$$ {}_g[\![v := ?]\!]^{M,\mathcal{E}}_h \text{ iff } h = g[\textit{the answer to life, the universe and everything} \mapsto 42] $$

On a side note, this example poses some interesting questions about 'negative' side effects. Under our current definition, setting the above mentioned variable to 42 registers as a side effect, but in a somewhat strange fashion. After all $v := ?$ is a single instruction and for ${}_g[\![\rho]\!]^M_h$ and ${}_g[\![\rho]\!]^{M,\mathcal{E}}_{h'}$, $\mathcal{S}^M_g(\rho) = \delta(h', h)$. There will actually be two differences between valuations h' and h here: the actual evaluation updates variable v, whereas the expected evaluation leaves v alone but does update the variable *the answer to life, the universe and everything*. Both variables will show up in the set of side effects, both with the assignment the actual evaluation has assigned to them.

This fails to capture what has actually happened here: after all, not only did an unexpected change to the initial valuation happen (a 'regular' side effect), but an expected change also did *not* happen (a 'negative' side effect). At least part of the information what should have happened is lost, namely the value the variable *the answer to life, the universe and everything* was supposed to get.[2] It is an open question if we should even allow these somewhat odd situations where the actual evaluation does something completely different than we expect, thereby generating a negative side effect. We leave this question, as well as the question how we should handle these situations if we do choose to allow them, for future work.

[2] Which is quite a shame, considering the trouble it cost to get it.

7 A classification of side effects

7.1 Introduction

In this chapter we will take a closer look at side effects in steering fragments. In particular, we will give a classification of side effects. This classification gives us a measure of the impact of a side effect.

As we have already mentioned in our introduction in Chapter 1, Bergstra has given an informal classification of side effects in [1]. Bergstra makes a distinction between steering instructions and working instructions. This distinction is based on a setting called Program Algebra (PGA). In PGA, there is no distinction between formulas and single instructions other than formulas, which is why the proposed distinction by Bergstra is meaningful in that setting. Every basic instruction a in PGA yields a Boolean reply upon execution and can therefore be made into a positive or negative test instruction $+a$ or $-a$. Naturally, this cannot be done in our setting of $\mathrm{DLA_f}$, so instead of giving an overview of Bergstra's paper, I will just present the major classes of side effects Bergstra distinguishes and what they come down to in our setting.

Bergstra's first class of side effects is what he calls 'trivial side effects'. By this he means side effects that can only be found in e.g. consequences for the length of the program or its running time. We are usually not interested in those kinds of side effects, which is exactly why Bergstra calls them trivial and why we would say that no side effects occur at all. An instruction that only returns a meaningful Boolean reply (that is, a Boolean reply that may differ depending on the valuation the instruction is evaluated in) is an instruction that only has trivial side effects. Examples of such instructions are the comparision instructions such as $(x = 2)$ or $(x \leq 2)$. These instructions can be turned into meaningful test instructions by prefixing them with a $+$ or $-$ symbol. We will return to this in our explanation of PGA in Chapter 8. In our terms, these kinds of instructions can only be formulas, occuring in steering fragments such as $?(x = 2)$ or $?(x \leq 2)$. To be precise, they can only be formulas that have the same actual and expected evaluation, and thus no side effects.

The above described situation, where only trivial side effects occur, is one extreme. The other extreme is when an instruction always yields the same Boolean reply, regardless of when it is executed. Bergstra says that in that case,

only 'trivial Boolean results' occur and that the instruction should be classified as a working instruction (that is, a single instruction not being a formula). In our setting this is also true with one notable exception: that of assignments. As we know, assignments always return true, so their Boolean result is trivial. Still, we allow them in formulas, too. If an instruction with trivial Boolean results occurs outside a formula, its only relevance would be its effect other than the Boolean reply, in which case you can hardly call that effect a side effect. If it occurs in a formula, however, the Boolean result — albeit trivial — does have relevance, so the effect other than the Boolean reply can indeed be called a side effect. This is exactly what happens in our setting.

What the classification between steering instructions and working instructions gives us in the end, is a recommendation on how to use a particular kind of instruction. Instructions such as comparision ($x \leq 2$), that only give a Boolean reply, have no meaning as a working instruction and therefore ideally should only occur in steering fragments. Other instructions such as assignment ($x := 2$) can be both steering instructions as well as working instructions and can thus occur both inside as well as outside steering fragments. Finally, instructions such as writing to the screen (`write x`) do not return a meaningful Boolean reply and should therefore ideally not occur in steering fragments.

7.2 Marginal side effects

7.2.1 Introduction

Having seen the base class of side effects, we can move on to the next level, that of *marginal side effects*. The intuition behind a marginal side effect is fairly simple: the side effect of a single instruction is marginal if the remainder of the execution of the program is unaffected by the occurrence of the side effect. The following program is a typical example of one where a marginal side effect occurs:

$$d\pi = d\pi_1; ?([x := x + 1]\top); y := 1$$

Here $d\pi_1$ can be any (deterministic) program. The side effect occurs in the test. However, since the variable x is no longer used in the remainder of the program (which only consists of the single instruction $y := 1$), the remainder of the program is unaffected by the occurrence of the side effect. Therefore, this side effect is marginal.

So what if x does occur in the remainder of the program, for example in this program:

$$d\pi = d\pi_1; ?([x := x + 1]\top); x := x + 1$$

This is a typical example of a program in which the occuring side effect is not marginal. The reason is that the assignment in the remainder of the program ($x := x + 1$) has a different effect on the variable x than when it would have had if the side effect had not occurred. For instance, for initial valuation g such that $g(x) = 1$ (and assuming x does not occur in π_1), the assignment maps x to 3. If the side effect had not occurred, it would have had a different effect on x (namely, it would have mapped it to 2).

Another typical example of a program in which an occuring side effect is not

marginal is our running example:

$$d\pi = d\pi_1; ?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} (x = 2)); y := 1$$

Here $d\pi_1$ can again be any deterministic program and the side effect occurs in the same place as in our first example. However, the test is now a complex test and in the second part of the test, x is used. Suppose the valuation after evaluation of $d\pi_1$ is f such that $f(x) = 1, f(y) = 2$. The second part of the test $(x = 2)$ will now give a different reply if a side effect does not occur in the first part (or if that side effect would have affected a different variable). As a result, the remainder of the program is affected by the side effect: it will be executed differently if a side effect occurs.

Perhaps the answer to the question if the side effect is marginal is less clear when the initial valuation in the previous example would not have been g with $g(x) = 1$, but for example with $g(x) = 42$. It is still the case that the variable x, that is affected by a side effect, is used again in the remainder of the program, but now it does not change the outcome of the (complex) test. Is that side effect still not marginal then? The same question can be posed about the following example:

$$d\pi = d\pi_1; ?([x := x + 1]\top); x := 42$$

Regardless of initial valuation g, at the end of this program (assuming $d\pi_1$ terminates), x will always be mapped to 42. So is the side effect in the test marginal or not? The answer can be found by checking if the remainder of the program is executed in the same way, or more formally: if the actual update of the remainder of the program is the same regardless of whether a side effect has occurred. In both our last examples, the answer to that last question is yes. After all, in the first example the test $x = 2$ will fail whether x has been incremented first or not, and in the second example x will always be mapped to 42, again regardless of the side effect that incremented x earlier. Therefore, the side effects in the discussed instructions are marginal.

7.2.2 Marginal side effects in single instructions

Although the intuition of marginal side effects should be clear enough by now, formally defining it is tricky because we have to define precisely what the remainder of a (deterministic) program $d\pi$ given a single instruction ρ and an initial valuation g is. Before we can define that, we also need to know the *history* of that same program given single instruction ρ, which is loosely described as those (single or basic) instructions that have already been evaluated when ρ is about to get evaluated.

In what follows we are going to assume that in a certain deterministic program $d\pi$ a single instruction ρ occurs that is causing a side effect. Furthermore, we are going to use that given initial valuation g, any deterministic program has a unique canonical form that has the same behavior (see Proposition 4 in Chapter 4). Defining the history and remainder of a deterministic program is straight-forward if that program is in canonical form. Also, we can actually immediately give a more general definition than what we need here, namely the history and remainder of a deterministic program given a basic instruction. This extra generality will come in handy later on.

Definition 21. *Let $d\pi$ be a deterministic program in canonical form. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$. Let ϖ be a basic instruction occuring in $d\pi$, that is, $d\pi$ is of the form $d\pi_1; \varpi; d\pi_2$, with $d\pi_1$ and $d\pi_2$ being possibly empty deterministic programs in canonical form. The* ***history*** *of program $d\pi$ given basic instruction ϖ is defined as:*

$$\mathcal{H}^M_g(d\pi, \varpi) = \begin{cases} ?\top & \text{if } d\pi_1 \text{ is empty} \\ d\pi_1 & \text{o.w.} \end{cases}$$

The ***remainder*** *of program $d\pi$ given basic instruction ϖ is defined as:*

$$\mathcal{R}^M_g(d\pi, \varpi) = \begin{cases} ?\top & \text{if } d\pi_2 \text{ is empty} \\ d\pi_2 & \text{o.w.} \end{cases}$$

Using Proposition 4 the extension of the definitions of history and remainder of a program to all deterministic programs (not just the ones in canonical form) is trivial:

Definition 22. *Let $d\pi$ be a deterministic program. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]^M_h$. Furthermore, let $d\pi'$ be the deterministic program in canonical form as meant in Proposition 4. The* ***history*** *of program $d\pi$ given basic instruction ϖ is defined as:*

$$\mathcal{H}^M_g(d\pi, \varpi) = \mathcal{H}^M_g(d\pi', \varpi)$$

The ***remainder*** *of program $d\pi$ given basic instruction ϖ is defined as:*

$$\mathcal{R}^M_g(d\pi, \varpi) = \mathcal{R}^M_g(d\pi', \varpi)$$

With definitions for the history and the remainder of a program in hand, we can define marginal side effects. According to our intuition, a side effect should be marginal if the evaluation of the remainder of the program is the same regardless of whether the side effect occurred. We can tell if that is the case by evaluating the remainder of the program with two different valuations: one in which the single instruction in which the side effect occurs has been evaluated using the actual evaluation, and one in which is has been evaluated using the expected evaluation.[1] If the only difference between those two valuations is exactly the side effect that occurred in the single instruction, or if there is no difference between those two valuations at all, then we can say that the evaluation of the remainder of the program has been the same. This is formally defined as follows:

Definition 23. *Let $d\pi$ be a deterministic program. Let model M and initial valuation g be given and let h_A be the valuation such that ${}_g[\![d\pi]\!]^M_{h_A}$. Let ρ be a single instruction in program $d\pi$ causing a side effect, that is, for ${}_g[\![\mathcal{H}^M_g(d\pi, \rho)]\!]^M_f$, $\mathcal{S}^M_f(\rho) \neq \emptyset$. Let f_A be the valuation such that ${}_f[\![\rho]\!]^M_{f_A}$ and let f_E be the valuation such that ${}_f[\![\rho]\!]^{M,\mathcal{E}}_{f_E}$. The side effect in ρ is marginal iff for ${}_{f_A}[\![\mathcal{R}^M_g(d\pi, \rho)]\!]^M_{h_A}$*

$$\exists h_E \text{ s.th. } {}_{f_E}[\![\mathcal{R}^M_g(d\pi, \rho)]\!]^{M,\mathcal{E}}_{h_E} \text{ and } \delta^M(h_E, h_A) = (\mathcal{S}^M_f(\rho) \text{ or } \emptyset)$$

[1] We now need to restrict ourselves again to single instructions because the expected evaluation is (currently) undefined for complex steering fragments.

So what happens here exactly? To show this, we return to the examples we have given earlier in this section. First, consider the program $d\pi = x := 1; ?([x := x+1]\top); y := 1$, with initial valuation g such that $g(x) = g(y) = 0$. We can observe that $d\pi$ is in canonical form. In this program, a side effect occurs in the single instruction $\rho = ?([x := x + 1]\top)$. So is this side effect marginal or not? Here we have the following:

$$\begin{aligned}
\mathcal{H}_g^M(d\pi, \rho) &= (x := 1) \\
\mathcal{R}_g^M(d\pi, \rho) &= (y := 1) \\
f &= g[x \mapsto 1, y \mapsto 0] \\
f_A &= f[x \mapsto 2, y \mapsto 0] \\
f_E &= f[x \mapsto 1, y \mapsto 0] \\
h_A &= f_A[x \mapsto 2, y \mapsto 1] \\
h_E &= f_E[x \mapsto 1, y \mapsto 1]
\end{aligned}$$

As we can see, the valuations f and f_E are the same. Using our current definition of the expected evaluation, this will always be the case, so we could just use valuation f here. However, as I have said in Section 6.5 of Chapter 6, I want to keep generality in the definitions of side effects. We might want to change the definition of the expected evaluation in the future or add new instructions or connectives that do modify the initial valuation. Therefore, we use valuation f_E, the resulting valuation after evaluating the single instruction ρ with the expected evaluation.

To determine if the side effects are marginal, we have to ask ourselves if

$$\delta^M(h_E, h_A) = \mathcal{S}_f^M(\rho) \text{ or } \emptyset$$

We know how to calculate the set of side effects; it is $\{x \mapsto 2\}$. In this case, $\delta^M(h_E, h_A)$ is $\{x \mapsto 2\}$ too, so the side effect occurring in ρ is marginal, which is what we want. We can also clearly see in this case that it is no coincidence that we are testing $\delta^M(h_E, h_A)$ and not $\delta^M(h_A, h_E)$: we need the valuation that is the result of evaluating the single instruction using the actual evaluation in order to properly compare this with the set of side effects.

We can now take a look at an example in which the side effect should not be marginal. Consider the program $d\pi = x := 1; ?([x := x + 1]\top); x := x + 1$, with initial valuation g such that $g(x) = 0$. This program is in canonical form too and the side effect occurs in the same single instruction ρ. This time we get the following:

$$\begin{aligned}
\mathcal{H}_g^M(d\pi, \rho) &= (x := 1) \\
\mathcal{R}_g^M(d\pi, \rho) &= (x := x + 1) \\
f &= g[x \mapsto 1] \\
f_A &= f[x \mapsto 2] \\
f_E &= f[x \mapsto 1] \\
h_A &= f_A[x \mapsto 3] \\
h_E &= f_E[x \mapsto 1]
\end{aligned}$$

We have the same set of side effects: $\{x \mapsto 2\}$. However, $\delta^M(h_E, h_A)$ now is $\{x \mapsto 3\}$. Therefore the side effect is not marginal, which is again what we would expect.

We have given a third example which closely resembles the ones we have discussed above, namely $d\pi = x := 1; ?([x := x + 1]\top); x := 42$. If we take the same initial valuation g as above, everything except the remainder of the program given ρ will be the same:

$$\begin{aligned}
\mathcal{H}_g^M(d\pi, \rho) &= (x := 1) \\
\mathcal{R}_g^M(d\pi, \rho) &= (x := 42) \\
f &= g[x \mapsto 1] \\
f_A &= f[x \mapsto 2] \\
f_E &= f[x \mapsto 1] \\
h_A &= f_A[x \mapsto 42] \\
h_E &= f_E[x \mapsto 42]
\end{aligned}$$

With this example we can see why our definition of marginal side effects allows the difference between h_A and h_E to be $\emptyset$, too. We have seen before that in situations like these, the side effects should be marginal, and by allowing the difference to be $\emptyset$, that indeed is the case.

7.2.3 Marginal side effects caused by primitive formulas

As we have seen, our current definition of marginal side effects is capable of determining whether a side effect occurring in a single instruction is marginal or not. We still have to define marginal side effects for basic instructions. In particular, we need to have a definition for the situation in which a primitive formula in a complex test causes a side effect[2] and in that same test, the variable affected by that side effect is used again, such as in the following program: $d\pi = d\pi_1; ?([x := x + 1]\top \mathbin{\diamond\!\!\wedge} (x = 2)); y := 1$. In order to define how to determine if a side effect is marginal or not in these situations, we need to extend our definitions of the history and remainder of a program such that it not only works given a single instruction, but also given a primitive formula. Before we can give that definition, we first need to define the history and remainder of a compound formula given a primitive formula. We are once again only interested in those two concepts if the primitive formula φ gets evaluated.

To get an idea of what the history and the remainder of a compound formula given a primitive formula should be, consider the following example:

$$\begin{aligned}
\varphi &= [x := 6]\top \\
\phi &= \neg\varphi \mathbin{\diamond\!\!\vee} (x \leq 10) \\
&= \neg([x := 6]\top) \mathbin{\diamond\!\!\vee} (x \leq 10)
\end{aligned}$$

In this example, the history of ϕ given φ and given model M and initial valuation g is empty. The remainder, however, is not:

$$\mathcal{R}_g(\phi, \varphi) = x \leq 10$$

[2] We say that a primitive formula causes a side effect here because a side effect cannot occur *in* a primitive formula. It can, however, occur in a single or basic instruction which tests that formula.

Notice that this remainder should be empty if $\neg\varphi$ would have been true.

The history of a formula of course is not always empty. To illustrate that, we will first introduce a notational convention.

Notation. *We will write $\phi(\underline{\varphi})$ to refer to the primitive formula φ occurring in formula ϕ at a specific position.*

As an example of this, compare the formulas $\phi_1(\underline{\varphi}) = \underline{\varphi} \mathbin{\diamondsuit\!\!\wedge} \varphi$ and $\phi_2(\underline{\varphi}) = \varphi \mathbin{\diamondsuit\!\!\wedge} \underline{\varphi}$. The difference between the formulas $\phi_1(\underline{\varphi})$ and $\phi_2(\underline{\varphi})$ is in the instance of primitive formula φ we are referring to.

Let $\varphi = [x := 6]\top$ and $\phi(\underline{\varphi})$ as in the example above. Now consider the following example:

$$\psi(\underline{\varphi}) = (x = 2 \mathbin{\diamondsuit\!\!\wedge} \phi(\underline{\varphi}))$$

Here the history of ψ given $\underline{\varphi}$ and given model M and initial valuation g such that $g(x) = 2$ is not empty:

$$\mathcal{H}_g^M(\psi, \underline{\varphi}) = (x = 2)$$

Now that we have given an intuition what the history and remainder of a formula given a primitive formula and an initial valuation are going to be, we can move on to giving the actual definitions. In what follows we will assume that the ϕ in $\mathcal{H}_f(\phi, \underline{\varphi})$ is in normal form and that the specific primitive formula $\underline{\varphi}$ actually appears exactly once in formula $\phi(\underline{\varphi})$ (although other instances of φ may occur in the formula). $\phi(\underline{\varphi})$ can take the following forms:

$$\varphi(\underline{\varphi}), \quad \neg\varphi(\underline{\varphi}), \quad \phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\vee} \phi_2, \quad \phi_1 \mathbin{\diamondsuit\!\!\vee} \phi_2(\underline{\varphi}), \quad \phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\wedge} \phi_2, \quad \phi_1 \mathbin{\diamondsuit\!\!\wedge} \phi_2(\underline{\varphi})$$

Here $\varphi(\underline{\varphi})$ is the same as φ. For each of these forms, we will have to define how the history and the remainder is calculated.

Definition 24. *Let ϕ be a formula of one of the above forms. Let model M and initial valuation g be given. Let $\underline{\varphi}$ be a primitive formula occurring in ϕ such that $\underline{\varphi}$ gets evaluated during the evaluation of ϕ given initial valuation g. The **history** of formula ϕ given primitive formula $\underline{\varphi}$ is defined as:*

$$\begin{aligned}
\mathcal{H}_g^M(\varphi(\underline{\varphi}), \underline{\varphi}) &= \top \\
\mathcal{H}_g^M(\neg\varphi(\underline{\varphi}), \underline{\varphi}) &= \top \\
\mathcal{H}_g^M(\phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\vee} \phi_2, \underline{\varphi}) &= \mathcal{H}_g^M(\phi_1(\underline{\varphi}), \underline{\varphi}) \\
\mathcal{H}_g^M(\phi_1 \mathbin{\diamondsuit\!\!\vee} \phi_2(\underline{\varphi}), \underline{\varphi}) &= \phi_1 \mathbin{\diamondsuit\!\!\vee} \mathcal{H}_g^M(\phi_2(\underline{\varphi}), \underline{\varphi}) \\
\mathcal{H}_g^M(\phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\wedge} \phi_2, \underline{\varphi}) &= \mathcal{H}_g^M(\phi_1(\underline{\varphi}), \underline{\varphi}) \\
\mathcal{H}_g^M(\phi_1 \mathbin{\diamondsuit\!\!\wedge} \phi_2(\underline{\varphi}), \underline{\varphi}) &= \phi_1 \mathbin{\diamondsuit\!\!\wedge} \mathcal{H}_g^M(\phi_2(\underline{\varphi}), \underline{\varphi})
\end{aligned}$$

*The **remainder** of formula ϕ given primitive formula $\underline{\varphi}$ is defined as:*

$$\begin{aligned}
\mathcal{R}_g^M(\varphi(\underline{\varphi}), \underline{\varphi}) &= \top \\
\mathcal{R}_g^M(\neg\varphi(\underline{\varphi}), \underline{\varphi}) &= \top \\
\mathcal{R}_g^M(\phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\vee} \phi_2, \underline{\varphi}) &= \mathcal{R}_g^M(\phi_1(\underline{\varphi}), \underline{\varphi}) \mathbin{\diamondsuit\!\!\vee} \phi_2 \\
\mathcal{R}_g^M(\phi_1 \mathbin{\diamondsuit\!\!\vee} \phi_2(\underline{\varphi}), \underline{\varphi}) &= \mathcal{R}_g^M(\phi_2(\underline{\varphi}), \underline{\varphi}) \\
\mathcal{R}_g^M(\phi_1(\underline{\varphi}) \mathbin{\diamondsuit\!\!\wedge} \phi_2, \underline{\varphi}) &= \mathcal{R}_g^M(\phi_1(\underline{\varphi}), \underline{\varphi}) \mathbin{\diamondsuit\!\!\wedge} \phi_2 \\
\mathcal{R}_g^M(\phi_1 \mathbin{\diamondsuit\!\!\wedge} \phi_2(\underline{\varphi}), \underline{\varphi}) &= \mathcal{R}_g^M(\phi_2(\underline{\varphi}), \underline{\varphi})
\end{aligned}$$

The reason we are only interested in the history and remainder of a primitive formula if that formula is actually evaluated, is straight-forward: we use these definitions to calculate the side effects caused by that primitive formula and those side effects only exist if the primitive formula is evaluated. As straightforward as this is, the restriction is an important one. Because we know that $\underline{\varphi}$ gets evaluated (not be be confused with 'yielding true'), we do not have to take potentially troublesome formulas into account such as $\bot \owedge \underline{\varphi}$.

The above definitions make the history and remainder of a formula given a primitive formula, partial functions. To see in which situations the history and remainder are defined and for which they are not, consider the following formula:

$$\phi = (x = 5 \owedge [x := x + 1]\top) \ovee [x := x + 2]\top$$

Now assume we want to know the history of ϕ given $\varphi = [x := x + 1]\top$. This history $\mathcal{H}_g^M(\phi(\underline{\varphi}), \underline{\varphi})$ is only defined if $[x := x + 1]\top$ gets evaluated, which in turn only is the case if we have a initial valuation g such that $g(x) = 5$. For all initial valuations g' such that $g(x) \neq 5$, the history of ϕ given φ is undefined. If we would be interested in the history of ϕ given $\varphi' = [x := x+2]\top$, the situation would be reversed: in that case the history $\mathcal{H}_g^M(\phi(\underline{\varphi'}), \underline{\varphi'})$ is only undefined with initial valuation g such that $g(x) = 5$.

That the history (and the remainder) is undefined in these cases is not problematic because as said, we are going to use these definitions to check if the side effects caused by $\underline{\varphi}$ are marginal and $\underline{\varphi}$ can only cause side effects if it gets evaluated.

Using these definitions, we can move on to define the history and remainder of a program given a primitive formula:

Definition 25. *Let $d\pi$ be a deterministic program in canonical form. Let model M and initial valuation g be given and let h be the valuation such that ${}_g[\![d\pi]\!]_h^M$. Let $?\phi$ be a test occurring in program $d\pi$, where ϕ is a formula in normal form. Finally, let $\underline{\varphi}$ be a primitive formula occuring in ϕ such that $\underline{\varphi}$ gets evaluated during the evaluation of ϕ given initial valuation g. The **history** of program $d\pi$ given primitive formula $\underline{\varphi}$ is, for ${}_g[\![?\mathcal{H}_g^M(d\pi, ?\phi)]\!]_f^M$, defined as:*

$$\mathcal{H}_g^M(d\pi, \underline{\varphi}) = \mathcal{H}_g^M(d\pi, ?\phi); ?\mathcal{H}_f^M(\phi(\underline{\varphi}), \underline{\varphi})$$

*The **remainder** of program $d\pi$ given primitive formula $\underline{\varphi}$ is defined as:*

$$\mathcal{R}_g^M(d\pi, \underline{\varphi}) = ?\mathcal{R}_f^M(\phi(\underline{\varphi}), \underline{\varphi}); \mathcal{R}_g^M(d\pi, ?\phi)$$

The final step is to give a definition to determine if a side effect occurring in a primitive formula is marginal. Given the above, this definition should not be surprising:

Definition 26. *Let $d\pi$ be a deterministic program. Let model M and initial valuation g be given and let h_A be the valuation such that ${}_g[\![d\pi]\!]_{h_A}^M$. Let $\underline{\varphi}$ be a primitive formula in program $d\pi$ causing one of the side effects of $d\pi$. Let f be the valuation such that ${}_g[\![\mathcal{H}_g^M(d\pi, \underline{\varphi})]\!]_f^M$. Let f_A be the valuation such that ${}_f[\![?\varphi]\!]_{f_A}^M$ or ${}_f[\![?\neg\varphi]\!]_{f_A}^M$ and let f_E be the valuation such that ${}_f[\![?\varphi]\!]_{f_E}^{M,\mathcal{E}}$ or*

${}_f[\![?\neg\varphi]\!]^{M,\mathcal{E}}_{f_E}$.[3] *The side effect caused by* $\underline{\varphi}$ *is marginal iff for* ${}_{f_A}[\![\mathcal{R}^M_g(d\pi,\underline{\varphi})]\!]^M_{h_A}$

$$\exists h_E \text{ s.th. } {}_{f_E}[\![?\mathcal{R}^M_g(d\pi,\underline{\varphi})]\!]^{M,\mathcal{E}}_{h_E} \text{ and } \delta^M(h_E,h) = (\mathcal{S}^M_f(?\underline{\varphi}) \text{ or } \emptyset)$$

To show how this works, we return to the example given in the beginning of this section: $d\pi = x := 1; ?([x := x+1]\top \between (x = 2)); y := 1$, with initial valuation g such that $g(x) = g(y) = 0$. Here the primitive formula $\varphi = [x := x + 1]\top$ causes a side effect. We can now use our definition to find out if that side effect is marginal. For that, we first need the history of $d\pi$ given primitive formula $\underline{\varphi}$. To calculate $\mathcal{H}^M_g(d\pi,\underline{\varphi})$, we first observe that ϕ is in normal form. This gives us a go to use Definition 25. This definition tells us to first calculate valuation f, which we get by evaluating ${}_g[\![\mathcal{H}^M_g(d\pi,?\phi)]\!]^M_f$. Here $?\phi$ is a basic instruction, so we can use Definition 22 to calculate it. We have seen before how that evaluates:

$$\mathcal{H}^M_g(d\pi,?\phi) = (x := 1)$$

Thus we get ${}_g[\![x := 1]\!]^M_f$, so $f = g[x \mapsto 1, y \mapsto 0]$.

All we need to do now to get the history we are looking for, is the history of formula ϕ given primitive formula $\underline{\varphi}$: $\mathcal{H}^M_f(\phi(\underline{\varphi}),\underline{\varphi})$. We can use Definition 24 here and are in the situation where $\phi(\underline{\varphi}) = \phi_1(\underline{\varphi}) \between \phi_2$. Here $\phi_1 = \varphi$ and $\phi_2 = (x = 2)$, so as history we get:

$$\begin{aligned}\mathcal{H}^M_f(\phi(\underline{\varphi}),\underline{\varphi}) &= \mathcal{H}^M_f(\phi_1(\underline{\varphi}) \between \phi_2,\underline{\varphi})\\ &= \mathcal{H}^M_f(\phi_1(\underline{\varphi}),\underline{\varphi})\\ &= \mathcal{H}^M_f(\varphi(\underline{\varphi})),\underline{\varphi})\\ &= \top\end{aligned}$$

Thus, the history of program $d\pi$ given primitive formula $\underline{\varphi}$ is:

$$\begin{aligned}\mathcal{H}^M_g(d\pi,\underline{\varphi}) &= \mathcal{H}^M_g(d\pi,?\phi);?\mathcal{H}^M_f(\phi(\underline{\varphi}),\underline{\varphi})\\ &= (x := 1);?\top\end{aligned}$$

With the information above we can also immediately calculate the remainder of formula ϕ given primitive formula $\underline{\varphi}$:

$$\begin{aligned}\mathcal{R}^M_f(\phi(\underline{\varphi}),\underline{\varphi}) &= \mathcal{R}^M_f(\phi_1(\underline{\varphi}) \between \phi_2,\underline{\varphi})\\ &= \mathcal{R}^M_f(\phi_1(\underline{\varphi}),\underline{\varphi}) \between \phi_2\\ &= \mathcal{R}^M_f(\varphi(\underline{\varphi}),\underline{\varphi}) \between \phi_2\\ &= \top \between (x = 2)\end{aligned}$$

Then all we need to determine the remainder of program $d\pi$ given primitive formula $\underline{\varphi}$ is the remainder of program $d\pi$ given basic instruction $?\phi$. To see how this evaluates, see the previous section. We can use Definition 22 for this again and get:

$$\begin{aligned}f_A &= f[x \mapsto 2, y \mapsto 0]\\ \mathcal{R}^M_{f_A}(d\pi,\underline{\varphi}) &= (y := 1)\end{aligned}$$

[3]This distinction is necessary because we can only evaluate a test if its argument yields true. $M \models_f \varphi$ might actually yield false if φ is part of a larger formula ϕ that despite that yields true, such as $\phi = \varphi \overline{\vee} \phi_1$ such that $M \models_{f_A} \phi_1$. Thus, we need either φ or $\neg\varphi$.

So the remainder of program $d\pi$ given primitive formula $\underline{\varphi}$ is:

$$\begin{aligned}\mathcal{R}_g^M(d\pi, \underline{\varphi}) &= ?\mathcal{R}_f^M(\phi(\underline{\varphi}), \underline{\varphi}); \mathcal{R}_{f_A}^M(d\pi, ?\phi) \\ &= ?(\top \between (x = 2)); (y := 1)\end{aligned}$$

Now that we have the history and the remainder of $d\pi$ given $\underline{\varphi}$, we can finally determine if the side effect occurring in $\underline{\varphi}$ is marginal. To quickly recap, we have:

$$\begin{aligned}\mathcal{H}_g^M(d\pi, \underline{\varphi}) &= (x := 1); ?\top \\ \mathcal{R}_g^M(d\pi, \underline{\varphi}) &= ?(\top \between (x = 2)); (y := 1) \\ f &= g[x \mapsto 1, y \mapsto 0] \\ f_A &= f[x \mapsto 2, y \mapsto 0] \\ f_E &= f[x \mapsto 1, y \mapsto 0] \\ h_A &= f_A[x \mapsto 2, y \mapsto 1] \\ h_E &\text{ does not exist}\end{aligned}$$

Here we have an example where we do not even have to determine if $\delta^M(h_E, h_A)$ is the same as $\mathcal{S}_f^M(?\underline{\varphi})$, because there is no valuation h_E such that

$${}_{f_E}[\![\mathcal{R}_g^M(d\pi, \underline{\varphi})]\!]_{h_E}^{M,\mathcal{E}}$$

This is because for valuation f_E the test $?(\top \between (x = 2))$ will fail. Therefore, the side effect in $\underline{\varphi}$ is 'automatically' not marginal, which is indeed what we wanted.

7.3 Other classes of side effects

There are two more classes of side effects that I want to discuss. The first is the class *detectible side effects*. According to Bergstra, a side effect in an instruction is detectible if the fact that that side effect has occured can be measured by means of a steering fragment containing that instruction [1]. This is the most general class of side effects: in my terms, any difference between the actual and the expected evaluation of a single instruction is a detectible side effect.

The presence of detectible side effects suggests there are non-detectible side effects as well. This can indeed be the case. A side effect is undetectible if the evaluation of a (single) instruction causing a side effect would normally change the program state, but because of the specific initial valuation, it does not. As a simple example, consider the single instruction $?([v := 1]\top)$. Under any initial valuation g this would change the program state and cause a side effect, with one exception: namely if $g(v) = 1$. We can formally define this as follows:

Definition 27. *Let ρ be a single instruction in model M under initial valuation g, updating the valuation of a variable v.*[4] *Furthermore, let $\mathcal{S}_g^M(\rho) = \emptyset$. ρ contains an **undetectible side effect** iff for h such that $h(v) \neq g(v)$:*

$$\mathcal{S}_h^M(\rho) \neq \emptyset$$

[4] In $\mathrm{DLA_f}$, this would mean that ρ either is $v := t$ or $?[v := t]\top$.

It remains to be seen whether these non-detectible side effects are worth our attention. After all, not being able to detect side effects suggests that the presence of the side effects does not make much difference, in any case not to the further execution of the program. Possible exceptions to this are the execution speed or the efficiency of the program, especially if there are a lot of undetectible side effects.

In contrast to non-detectible side effects, marginal side effects can potentially be very useful because they can occur far more often. Like non-detectible side effects, they are a measure of the impact of a side effect. If a side effect is marginal, that means that the rest of the program is unaffected by it and therefore, the side effect is essentially pretty harmless. One could at this point imagine a claim that a program in which only marginal side effects occur can be considered a well-written program, whereas a program in which non-marginal side effects occur is one that should probably be rewritten to avoid unexpected behavior. We will leave further investigation of this claim for future work, however.

8 A case study: Program Algebra

In Chapter 6, I presented the system I will be using for the treatment of side effects. In this chapter I will provide a case study to see my system in action. For this, we will use *Program Algebra* (PGA) [3]. Since PGA is a basic framework for sequential programming, it provides an ideal case study for our treatment of side effects. By showing how side effects are determined in the very general setting of PGA, we are essentially showing how they are dealt with on a host of different, more specific programming languages.

I will first summarize PGA and explain how we can use it. Next, some extensions necessary for our purpose will be presented. Finally, I will present some examples to see in full how my system deals with side effects.

8.1 Program Algebra

8.1.1 Basics of PGA

PGA is built from a set A of basic instructions (not to be confused with the $\mathrm{DLA_f}$-notion by the same name), which are regarded as indivisible units. Basic instructions always provide a Boolean reply, which may be used for program control (i.e. in steering fragments). There are two composition constructs: concatenation and repetition. If X and Y are programs, then so is their concatenation $X;Y$ and its repetition X^ω. PGA has the following primitive instructions:

- **Basic instruction** Basic instructions are typically notated as `a,b,...`. As said they generate a Boolean value. Especially important for our purpose is that their associated behavior may modify a (program) state.
- **Termination instruction** This instruction, notated as !, terminates the program.
- **Test instruction** Test instructions come in two flavours: the positive test instruction, notated as $+a$ (where a is a basic instruction), and its negative counterpart, $-a$. For the positive test instruction, a is evaluated and if it yields `true`, all remaining instructions are executed. If it yields `false`, the next instruction is skipped and evaluation continues with the

instruction after that. For the negative test instruction, this is the other way around.

- **Forward jump instruction** A jump instruction, notated as $\#k$ where k can be any natural number. This instruction prescribes a jump to k instructions from the current one. If $k = 0$, the program jumps to the same instruction and inaction occurs. If $k = 1$, the program jumps to the next instruction (so this is essentially useless). If $k = 2$, the next instruction is skipped and the program proceeds with the one after that, and so on.

If two programs execute identical sequences of instructions, *instruction sequence congruence* holds between them. This can be axiomatized by the following four axioms:

$$(X;Y);Z = X;(Y;Z) \tag{PGA1}$$
$$(X^n)^\omega = X^\omega \tag{PGA2}$$
$$X^\omega;Y = X^\omega \tag{PGA3}$$
$$(X;Y)^\omega = X;(Y;X)^\omega \tag{PGA4}$$

The *first canonical form* of a PGA program is then defined to be a PGA program which is in one of the following two forms:

1. X not containing a repetition

2. $X;Y^\omega$, with both X and Y not containing a repetition

Any PGA program can be rewritten into a first canonical form using the above four equations. The next four axiom schemes for PGA deal with the simplification of chained jumps:

$$\#n+1;u_1;\ldots;u_n;\#0 = \#0;u_1;\ldots;u_n;\#0 \tag{PGA5}$$
$$\#n+1;u_1;\ldots;u_n;\#m = \#n+m+1;u_1;\ldots;u_n;\#m \tag{PGA6}$$
$$(\#n+k+1;u_1;\ldots;u_n)^\omega = (\#k;u_1;\ldots;u_n)^\omega \tag{PGA7}$$
$$X = u_1;\ldots;u_n;(v_1;\ldots;v_{m+1})^\omega \rightarrow \#n+m+k+2;X = \#n+k+1;X \tag{PGA8}$$

Programs are considered to be *structurally congruent* if they can be proven equal using the axioms PGA1-8.

The *second canonical form* of a PGA program is defined to be a PGA program in first canonical form for which additionally the following holds:

1. There are no chained jumps

2. Counters used for a jump into the repeating part of the expression are as short as possible

Each PGA expression can be rewritten into a shortest structurally equivalent second canonical form using the above eight equations [3].

8.1.2 Behavior extraction

The previous section describes the forms a PGA program can take. In this section I will explain the behavioral semantics defined in [3]. The process of determining the behavior of a PGA program given its instructions is called *behavior extraction*. The behavioral semantics itself is based on thread algebra, TA in short.

Like PGA, TA has a set A of basic instructions, which in this setting are referred to as actions. Furthermore, it has the following two constants and two composition mechanisms:

- **Termination** This is notated as `S` (for Stop) and terminates the behavior.
- **Divergent behavior** This is notated as `D` (for Divergence). Divergence (or inaction) means there no longer is active behavior. For instance, infinite jump loops cause divergent behavior since the program only makes jumps and does not perform any actions.
- **Postconditional composition** This is notated as $P \trianglelefteq a \trianglerighteq Q$ and means that first a is executed; if its reply is `true` then the behavior proceeds with P, otherwise it proceeds with Q.
- **Action prefix** This is notated as $a \circ P$ and is a shorthand for $P \trianglelefteq a \trianglerighteq P$: regardless of the reply of a, the behavior will proceed with P.

As said, behavior extraction determines the behavior of a PGA program given its instructions. For that, the behavior extraction operator, notated as $|\_|$, is defined. If a program ends without an explicit termination instruction, it is defined to end in inaction by the following equation:

$$|X| = |X; (\#0)^\omega| \tag{8.1}$$

A termination instruction followed by other instructions ends in termination and nothing else, which is defined by the following equation:

$$|!; X| = \mathrm{S} \tag{8.2}$$

Behavior extraction is further defined by the following equations dealing with the composition mechanisms:

$$|a; X| = a \circ |X| \tag{8.3}$$
$$|+a; u; X| = |u; X| \trianglelefteq a \trianglerighteq |X| \tag{8.4}$$
$$|-a; u; X| = |X| \trianglelefteq a \trianglerighteq |u; X| \tag{8.5}$$

The jump instruction requires a set of equations as well. The first equation defines that a jump instruction which is jumping to itself leads to inaction. The second and third define how a jump instruction can skip subsequent instructions.

$$|\#0; X| = \mathrm{D} \tag{8.6}$$
$$|\#1; X| = |X| \tag{8.7}$$
$$|\#k + 2; u; X| = |\#k + 1; X| \tag{8.8}$$

8.1.3 Extensions of PGA

PGA is a most basic framework [18]. However, there are many extensions that introduce more 'advanced' programming features such as goto's and backward jump instructions. Via projections, each of these extensions can be projected to PGA in such a way that the resulting PGA-program is behaviorally equivalent to the original program. Examples of such extensions are *PGLB*, in which PGA is extended with a backward jump instruction ($\backslash\#k$) and $PGLB_g$, in which PGLB is further extended with a label catch instruction ($L\sigma$) and an absolute goto instruction ($\#\#L\sigma$).

Of particular interest for our purpose is the extension of PGA with the *unit instruction operator* (PGA_u), introduced in [18]. The idea of the unit instruction operator, notated as $\mathbf{u}(\_)$, is to wrap a sequence of instructions into a single unit of length 1. That way, a more flexible style of PGA-programming is possible. In particular, programs of the form

```
if a then {
    b, c, d
} else {
    f, g, h
}
```

now have a more intuitive translation: $+a; \mathbf{u}(b; c; d; \#4;); f; g; h$.[1] Because, thanks to the unit instruction operator, the instructions b, c, d and $\#4$ are viewed as a single instruction, the execution of those is skipped when a yields `false`.

8.2 Logical connectives in PGA

8.2.1 Introduction

As mentioned in Section 8.1, in PGA a lot of basic notations for assembly-like programming languages are defined, especially with its extension with unit instruction operators (PGA_u) [18]. However, one important basic notation is missing: that of complex tests, of the form `if(a and b) then c`. As we have seen, currently there are positive and negative test instructions in PGA, which can only test the Boolean reply of a single instruction. More complex constructions such as the one in the working example of Section 3.2 are however very common in programming practice and also appear in research papers such as [1], where they are referred to as complex steering fragments. This means that for our purpose, PGA will have to be extended to accommodate for complex steering fragments. I will do so below.

Atomic steering fragments (that is, steering fragments containing only one instruction) are already present in PGA in the form of the positive and negative test instruction ($+a$ and $-a$ respectively). If we were to extend this with complex steering fragments, an obvious notation would be $+\phi$ and $-\phi$. The question now is what forms ϕ can take and what it means to have such a complex test.

Since the instructions in the steering fragment need to produce a Boolean reply, the answer to the question above in my opinion should be that a complex

[1] The jump is necessary to prevent the instructions f, g and h from being executed when a yields true.

test can only be meaningful if all the instructions in the complex test may be used to determine the reply. It is not necessary that all instructions are always used to determine the reply: for instance when using short-circuit evaluation, in some situations not all components of a complex test have to be (and therefore are not) used. However, my claim here is that if a certain instruction is *never* necessary to determine the Boolean reply of the whole steering fragment, then is should not be in the steering fragment.

Currently, PGA has two composition constructs (composition and repetition). Neither of those define anything, however, about the Boolean value of multiple instructions. That is, the Boolean value of $\phi; \ldots; \psi$ and of ϕ^ω is undefined. The intuitive way to determine the Boolean reply of a sequence of instructions is via logical connections such as And ($\wedge$) and Or ($\vee$). However, these are not present yet in PGA. This means that I will have to introduce them in an extension of PGA_u, which we baptize $\mathrm{PGA}_{\mathrm{ul}}$.

Before I do so, however, I need to say something more about the type of And and Or I will be using. There are multiple flavours available:

- **Logical And / Or** These versions are notated as $\wedge$ and $\vee$, respectively. They use full evaluation and the order of evaluation is undefined.
- **Short-circuit Left And / Or** These versions are the ones we use in $\mathrm{DLA}_{\mathrm{f}}$ (see Chapter 6). They are notated as $\mathbin{\diamondsuit\!\!\wedge}$ and $\mathbin{\diamondsuit\!\!\vee}$. From here on I will refer to them as SCLAnd and SCLOr. They use short-circuit evaluation and are therefore not commutative. The left conjunct or disjunct is evaluated first. There naturally are right-hand versions as well, but I will not be using them.
- **Logical Left And / Or** These versions are a combination of the other two: they use full evaluation, but the left conjunct or disjunct is evaluated first. I will notate this as & and |, respectively and refer to them as LLAnd and LLOr. I will not discuss right-hand versions.

The latter two are interesting for our purpose, because they are very suitable to demonstrate side effects. However, since we currently only have SCLAnd and SCLOr at our disposal in $\mathrm{DLA}_{\mathrm{f}}$, I will concentrate on those connectives. Although LLAnd and LLOr can be added to both PGA and $\mathrm{DLA}_{\mathrm{f}}$, this would raise more questions than it answers, for instance with regard to the logic which would then be behind the system, which is why we leave it for future work.

The above connectives will almost always be used in combination with either a positive or a negative test. This will be written as $+(a \mathbin{\diamondsuit\!\!\wedge} b)$ (and similar for the negative test and the $\mathbin{\diamondsuit\!\!\vee}$ connective).

8.2.2 Implementation of SCLAnd and SCLOr

If I am to introduce the mentioned logical connectives in $\mathrm{PGA}_{\mathrm{ul}}$, I will have to be able to project this extention into PGA. Since the projection of PGA_u to PGA is already given in [18], it is sufficient to project $\mathrm{PGA}_{\mathrm{ul}}$ to PGA_u to show that the former can be projected to PGA. Below is a proposal of a projection of the SCLAnd ($\mathbin{\diamondsuit\!\!\wedge}$) connective from $\mathrm{PGA}_{\mathrm{ul}}$ to PGA_u, for $a, b \in A$:

$$\mathrm{pgaul2pgau}(+(a \mathbin{\diamondsuit\!\!\wedge} b)) = \mathbf{u}(+a; \mathbf{u}(+b; \#2); \#2) \tag{8.9}$$

To see why this projection works, consider the following example: suppose we have the sequence $+\phi; c; d$ with $\phi = a \mathbin{◌\!\!\wedge} b$. This means that if a and b are true, c and d will be executed. Otherwise, only d will be executed. In $\mathrm{PGA}_{\mathrm{ul}}$ this sequence would be $+(a \mathbin{◌\!\!\wedge} b); c; d$. The projection to PGA_u would then be $\mathbf{u}(+a;\mathbf{u}(+b; \#2); \#2); c; d$. If a is false, the execution skips the unit and executes the jump instruction, ending up executing d. If a is true, the unit is entered, starting with the test b. If b is false, the execution again arrives at the same jump as before, skipping c and executing d. If b is true, a different jump is executed which makes the program jump to c first and only then moves on to d, which is exactly the desired behaviour.

The entire projection is wrapped in a unit because, as we will see later, the SCLAnd and other operators we define here also are to be considered units. Therefore, a program sequence prior to (or after) the operators discussed here cannot jump into the execution of that operator. By wrapping the projection into a unit I ensure that cannot happen after the projection either.

For the SCLOr connective, the projection is a little easier. It looks like this, again for $a, b \in A$:

$$\mathrm{pgaul2pgau}(+(a \mathbin{◌\!\!\vee} b)) = \mathbf{u}(-a; +b) \tag{8.10}$$

To see why this projection works, consider the same example as above: $+\phi; c; d$, but now with $\phi = a \mathbin{◌\!\!\vee} b$. So, if a and / or b are true, c and d should be executed. If they are both false, only d should be executed. In $\mathrm{PGA}_{\mathrm{ul}}$ this looks like this: $+(a \mathbin{◌\!\!\vee} b); c; d$. The projection to PGA_u then is $\mathbf{u}(-a; +b); c; d$. So, if a is true, execution skips testing b and moves on directly to c. If a is false, b is tested first. If b is also false, execution skips c and d is executed. If b is true, c gets executed first: exactly the desired behaviour.

So far, we have only been considering programs of the form $+\phi; c; d$, that is, with a positive test. Of course, we also have the negative test instruction. For a negative test, the projection of SCLAnd resembles that of SCLOr. This comes as no surprise since SCLAnd and SCLOr are each other's dual. It looks like this, again for $a, b \in A$:

$$\mathrm{pgaul2pgau}(-(a \mathbin{◌\!\!\wedge} b)) = \mathbf{u}(+a; -b) \tag{8.11}$$

The projection of $\mathbin{◌\!\!\vee}$ for a negative test resembles the projection of $\mathbin{◌\!\!\wedge}$ for a positive test:

$$\mathrm{pgaul2pgau}(-(a \mathbin{◌\!\!\vee} b)) = \mathbf{u}(-a; \mathbf{u}(-b; \#2); \#2) \tag{8.12}$$

8.2.3 Complex Steering Fragments

The implementations in the previous section work for steering fragments containing a single logical connective (that is, with disjuncts or conjuncts $a, b \in A$). However, we also need to define what happens for larger complex steering fragments (for instance $a \mathbin{◌\!\!\wedge} (b \mathbin{◌\!\!\vee} c)$). In order to accommodate this, we need one more property for the $\mathbin{◌\!\!\wedge}$ and $\mathbin{◌\!\!\vee}$ operators in PGA: they have to be treated as units. If we do this, we can give a recursive definition for the projection, with as base cases the ones given in the previous sections.

In what follows, the formulas ϕ_1 and ϕ_2 can take the following form:

$$\phi ::= \top \mid a \in A \mid \neg\phi \mid \phi \mathbin{◌\!\!\wedge} \psi \mid \phi \mathbin{◌\!\!\vee} \psi \tag{8.13}$$

As we can see, this includes negation. For more on negation, see the next section. We get the following projections:

$$\begin{aligned}
&\text{pgaul2pgau}(+(\phi_1 \between \phi_2)) = \mathbf{u}(\text{pgaul2pgau}(+\phi_1); \mathbf{u}(\text{pgaul2pgau}(+\phi_2); \#2); \#2)\\
&\text{pgaul2pgau}(+(\phi_1 \barwedge \phi_2)) = \mathbf{u}(\text{pgaul2pgau}(-\phi_1); \text{pgaul2pgau}(+\phi_2))\\
&\text{pgaul2pgau}(-(\phi_1 \between \phi_2)) = \mathbf{u}(\text{pgaul2pgau}(+\phi_1); \text{pgaul2pgau}(-\phi_2))\\
&\text{pgaul2pgau}(-(\phi_1 \barwedge \phi_2)) = \mathbf{u}(\text{pgaul2pgau}(-\phi_1); \mathbf{u}(\text{pgaul2pgau}(-\phi_2); \#2); \#2)
\end{aligned}$$

This works as follows. Consider the example $+\phi; d; !$, with $\phi = a \between (b \between c)$. In PGA_{ul} this would be written as:

$$+(a \between (b \between c)); d; ! \tag{8.14}$$

We can use our new recursive definition of $\between$ and get:

$$\begin{aligned}
\text{pgaul2pgau}(+(a \between (b \between c)); d; !) = \mathbf{u}\big(&\text{pgaul2pgau}(+a);\\
&\mathbf{u}(\text{pgaul2pgau}(+(b \between c)); \#2);\\
&\#2\big); d; !
\end{aligned}$$

The projections left now are base cases of $+a$ and $+(b \between c)$, respectively. Thus, we get

$$\begin{aligned}
\text{pgaul2pgau}(+(a \between (b \between c)); d; !) &= \mathbf{u}\big(\text{pgaul2pgau}(+a);\\
&\qquad \mathbf{u}(\text{pgaul2pgau}(+(b \between c)); \#2);\\
&\qquad \#2\big); d; !\\
&= \mathbf{u}\big(+ a;\\
&\qquad \mathbf{u}(\mathbf{u}(+b; \mathbf{u}(+c; \#2); \#2); \#2);\\
&\qquad \#2\big); d; !
\end{aligned}$$

An interesting question is whether these projections make $\between$ an associative operator. To find out, we compare the above with the example $+\phi; d; !$ where this time $\phi = (a \between b) \between c$. We get:

$$\begin{aligned}
\text{pgaul2pgau}(+((a \between b) \between c); d; !) &= \mathbf{u}\big(\text{pgaul2pgau}(+(a \between b));\\
&\qquad \mathbf{u}(\text{pgaul2pgau}(+c); \#2);\\
&\qquad \#2\big); d; !\\
&= \mathbf{u}\big(\mathbf{u}(+a; \mathbf{u}(+b; \#2); \#2);\\
&\qquad \mathbf{u}(+c; \#2);\\
&\qquad \#2\big); d; !
\end{aligned}$$

We can use behavior extraction to check if these programs are behavioral equivalent. It turns out that both programs indeed have the same behavior:

$$((d \circ S \trianglelefteq c \trianglerighteq S) \trianglelefteq b \trianglerighteq S) \trianglelefteq a \trianglerighteq S$$

Thus, we can conclude that $\between$ is associative in PGA_{ul}, as we would expect given SCL7. We can analyze $\barwedge$ in a similar manner.

8.2.4 Negation

Now that we have the projections for positive and negative tests defined, we can turn our attention to one more operator that is common both in programming practice and in logic: negation. In PGA, negation is absent, so we need to define it here. Not all instructions or sequences of instructions can be negated: after all, there is no intuition for the meaning of the negation of a certain behavior. We can, however, negate basic instructions: by this we mean its Boolean reply changes value. Sequences of instructions consisting of the operators I have defined above can be negated as well, which I will write as $\neg\phi$. First, I define the following standard projection rules:

$$+(\neg\phi) = -\phi \tag{8.15}$$
$$-(\neg\phi) = +\phi \tag{8.16}$$
$$\neg\neg\phi = \phi \tag{8.17}$$

Now that we have this, we need to take a look at how negation interacts with the $\between$ and $\between$ connectives. In particular, we are interested in what happens if one or both of the instructions in such a connective are negated. For this, the De Morgan's laws will come in handy:

$$\neg(\phi_1 \between \phi_2) = \neg\phi_1 \between \neg\phi_2 \tag{8.18}$$
$$\neg(\phi_1 \between \phi_2) = \neg\phi_1 \between \neg\phi_2 \tag{8.19}$$

With the above equations in combination with the equations 8.15-8.17, we already have the projections for two possible cases (namely when no instructions are negated and when both instructions are negated). That leaves us two other cases for both $\between$ and $\between$: one in which the first instruction is negated, and one in which the other is. Below are the projections of these cases:

$$\begin{aligned}\mathrm{pgaul2pgau}(+(\neg\phi_1 \between \phi_2)) &= \mathrm{pgaul2pgau}(-(\phi_1 \between \neg\phi_2)) \\ &= \mathbf{u}(\mathrm{pgaul2pgau}(+\phi_1); \#3; \mathrm{pgaul2pgau}(+\phi_2))\end{aligned} \tag{8.20}$$

$$\begin{aligned}\mathrm{pgaul2pgau}(+(\phi_1 \between \neg\phi_2)) &= \mathrm{pgaul2pgau}(-(\neg\phi_1 \between \phi_2)) \\ &= \mathbf{u}(\mathrm{pgaul2pgau}(-\phi_1); \#3; \mathrm{pgaul2pgau}(-\phi_2))\end{aligned} \tag{8.21}$$

$$\begin{aligned}\mathrm{pgaul2pgau}(+(\neg\phi_1 \between \phi_2)) &= \mathrm{pgaul2pgau}(-(\phi_1 \between \neg\phi_2)) \\ &= \mathbf{u}(\mathrm{pgaul2pgau}(-\phi_1); \#2; \mathrm{pgaul2pgau}(+\phi_2))\end{aligned} \tag{8.22}$$

$$\begin{aligned}\mathrm{pgaul2pgau}(+(\phi_1 \between \neg\phi_2)) &= \mathrm{pgaul2pgau}(-(\neg\phi_1 \between \phi_2)) \\ &= \mathbf{u}(\mathrm{pgaul2pgau}(+\phi_1); \#2; \mathrm{pgaul2pgau}(-\phi_2))\end{aligned} \tag{8.23}$$

For more on the $\between$ and $\between$ connectives and the rules that apply to them, see the paper by Bergstra and Ponse on short-circuit logic [5] as well as Chapter 5.

8.2.5 Other instructions

In the previous subsections we have seen what the projections of the new logical connectives in $\mathrm{PGA}_{\mathrm{ul}}$ to $\mathrm{PGA}_{\mathrm{u}}$ look like. To complete the list of projections,

we have to define the projections for the 'regular' instructions, as well as how concatenation and repetition are projected. This is trivial, since these 'regular' instructions are the same in $\mathrm{PGA_{ul}}$ and $\mathrm{PGA_u}$. We get for $a \in A$ and $\mathrm{PGA_{ul}}$-programs X, Y

$$\begin{aligned}
\mathrm{pgaul2pgau}(a) &= a \\
\mathrm{pgaul2pgau}(+a) &= +a \\
\mathrm{pgaul2pgau}(-a) &= -a \\
\mathrm{pgaul2pgau}(!) &= ! \\
\mathrm{pgaul2pgau}(\#k) &= \#k \\
\mathrm{pgaul2pgau}(X;Y) &= \mathrm{pgaul2pgau}(X);\mathrm{pgaul2pgau}(Y) \\
\mathrm{pgaul2pgau}(X^\omega) &= (\mathrm{pgaul2pgau}(X))^\omega \\
\mathrm{pgaul2pgau}(\mathbf{u}(X)) &= \mathbf{u}(\mathrm{pgaul2pgau}(X))
\end{aligned}$$

8.3 Detecting side effects in PGA

In this section I will show how to detect side effects in a $\mathrm{PGA_{ul}}$ program using our treatment of side effects. In essence, all we have to do is translate the $\mathrm{PGA_{ul}}$ program to an equivalent $\mathrm{DLA_f}$-program, which can then be used to determine the side effects that occur.

To recap, we have the following operators in $\mathrm{PGA_{ul}}$ that have to be translated:

- Concatenation $(X;Y)$
- Repetition (X^ω)
- Unit instruction operator $(\mathbf{u}(\_))$
- Termination $(!)$
- Positive and negative tests $(+\phi, -\phi)$
- Only in tests: conjunction, disjunction and negation ($\phi_1 \mathbin{\diamondsuit\!\!\wedge} \phi_2$,$\phi_1 \mathbin{\diamondsuit\!\!\vee} \phi_2, \neg\phi$)

There are two notable differences between $\mathrm{PGA_{ul}}$ and $\mathrm{DLA_f}$. The first is that in $\mathrm{PGA_{ul}}$ a program unsuccessfully terminates unless explicitly instructed otherwise by the termination instruction, whereas in $\mathrm{DLA_f}$ the default is a successful termination. This is an issue that has to be addressed to properly translate $\mathrm{PGA_{ul}}$ to $\mathrm{DLA_f}$ and the best way to do this, is to add the termination instruction to $\mathrm{DLA_f}$. This illustrates the point I made in Section 6.5 in Chapter 6: the instructions I defined so far in $\mathrm{DLA_f}$ are by no means exhaustive and new instructions may have to be added to them. This can usually be done by simply defining the actual and expected evaluation of the new instruction.

The nature of the termination instruction requires us to do a little more than just that. After all, the termination instruction has a control element to it: just like for instance the test instruction it has an influence on which instructions are to be evaluated next. To be exact, *no* instructions are to be evaluated next when a termination instruction is encountered during evaluation of a program. Because of this, we have to slightly modify the concatenation operator

in $\mathrm{DLA_f}$ too when we introduce the termination instruction. We baptize the extension of $\mathrm{DLA_f}$ with the termination instruction $\mathrm{DLTA_f}$ (for Dynamic Logic with Termination and Assignment in Formulas).

The equation for the relational meaning of ! in a given model M and initial valuation g is straight-forward. Execution simply finishes with the same resulting valuation as the initial valuation:

$$ {}_g[\![!]\!]^M_h \text{ iff } g = h \tag{DLTA15}$$

The updated rule for concatenation has to express that when a termination instruction is encountered, nothing should be evaluated afterwards. We use a case distinction for this on the first instruction of a concatenation:

$$ {}_g[\![\varpi; d\pi]\!]^M_h \text{ iff } \begin{cases} g = h & \text{if } \varpi = ! \\ \exists f \text{ s.th. } {}_g[\![\varpi]\!]_f \text{ and } {}_f[\![d\pi]\!]^M_h & \text{o.w.} \end{cases} \tag{DLTA12}$$

We only define the termination instruction in the setting of deterministic programs here. This is sufficient because this is the only setting we are currently interested in. DLTA12 replaces QDL12, but keeps the associative character of concatenation intact:

$$ {}_g[\![(d\pi_0; d\pi_1); d\pi_2]\!]^M_h = {}_g[\![d\pi_0; (d\pi_1; d\pi_2)]\!]^M_h $$

The addition of the termination instruction allows us to easily express $\mathrm{PGA_{ul}}$-programs such as $+a; !; b$ in $\mathrm{DLTA_f}$. They would otherwise have caused a problem because there would have been no easy way to stop the evaluation of the program from continuing to evaluating b, which it of course is not supposed to do if a yields true.

The other notable difference between $\mathrm{PGA_{ul}}$ and $\mathrm{DLA_f}$ is that in the former, anything can be used as a basic instruction. That includes what we refer to in $\mathrm{DLA_f}$ as primitive formulas such as $x \leq 2$ or $t_1 = t_2$. In PGA the execution of an instruction always succeeds, even if the Boolean reply that it generates, is false. To model this in $\mathrm{DLTA_f}$, we have to add the primitive formulas φ to the set of instructions, as follows:

$$ \pi ::= \varphi \mid ! \mid v := t \mid ?\phi \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* $$

The relational meaning in M given initial valuation g for these new instructions is simply that they always succeed without modifying g:

$$ {}_g[\![\varphi]\!]^M_h \text{ iff } g = h $$

With the termination instruction and the formulas-as-instructions defined, we can take a first look at the mapping from $\mathrm{PGA_{ul}}$ to $\mathrm{DLTA_f}$. For this we define a translation function $f_t : \mathrm{PGA_{ul}} \to \mathrm{DLTA_f}$. We define this translation function for PGA programs in first or second canonical form only; this is sufficient because as we have seen, every PGA program can be rewritten to first and second canonical form.

First, we define the set A of basic instructions in PGA to be equal to the set of primitive formulas and single instructions, not being tests, in $\mathrm{DLA_f}$:

$$ A ::= \varphi \mid \rho^- $$

where ρ^- denotes the set of single instructions not being tests. In $\mathrm{DLA_f}$, this set only consists of the assignment instruction $v := t$.

For finite sequences of instructions with length $n = 1$, $a, b \in A$ and $k \in \mathbb{N}_0$, and ϕ a formula as meant in section 8.2.3, f_t is defined as follows:

$$\begin{aligned} f_t(a) &= a; ?\bot \\ f_t(+\phi) &= ?\phi; ?\bot \\ f_t(-\phi) &= ?\neg\phi; ?\bot \\ f_t(\#k) &= ?\bot \\ f_t(!) &= ! \\ f_t(\mathbf{u}(a_1; \ldots; a_k)) &= f_t(a_1; \ldots; a_k) \end{aligned}$$

Here we can clearly see what effect it has that $\mathrm{PGA_{ul}}$ has unsuccessful termination as its default. We have to explicitly introduce unsuccessful termination in $\mathrm{DLTA_f}$ by adding $?\bot$ (a test that always fails) at the end of every instruction. Furthermore, notice the unit instruction operator that here has length $n = 1$, but is transparent when it has to be translated and thus becomes a sequence of instructions with length k that is potentially larger than 1. Finally, notice that there is no need to translate possibly compound formulas ϕ. This is because formulas have the exact same syntax in $\mathrm{PGA_{ul}}$ and $\mathrm{DLTA_f}$.

Next, we can show the definition of f_t for finite sequences of instructions with length $n = m + 1$. For $a, b_1, \ldots, b_m \in A$, $k \in \mathbb{N}_0$ and ϕ a formula as meant in section 8.2.3, we have

$$\begin{aligned} f_t(a; b_1; \ldots; b_m) &= a; f_t(b_1; \ldots; b_m) \\ f_t(+\phi; b_1; \ldots; b_m) &= \begin{cases} (?\phi; f_t(b_1)) \cup (?\neg\phi; ?\bot) & \text{if } m{=}1 \\ (?\phi; f_t(b_1; \ldots; b_m)) \cup & \\ \quad (?\neg\phi; f_t(b_2; \ldots; b_m)) & \text{o.w.} \end{cases} \\ f_t(-\phi; b_1; \ldots; b_m) &= \begin{cases} (?\phi; ?\bot) \cup (?\neg\phi; f_t(b_1)) & \text{if } m{=}1 \\ (?\phi; f_t(b_2; \ldots; b_m)) \cup & \\ \quad (?\neg\phi; f_t(b_1; \ldots; b_m)) & \text{o.w.} \end{cases} \\ f_t(\#0; b_1; \ldots; b_m) &= ?\bot \\ f_t(\#1; b_1; \ldots; b_m) &= f_t(b_1; \ldots; b_m) \\ f_t(\#(2{+}k); b_1; \ldots; b_m) &= \begin{cases} f_t(b_{k+2}; \ldots; b_m) & \text{if } k + 2 < m \\ ?\bot & \text{o.w.} \end{cases} \\ f_t(!; b_1; \ldots; b_m) &= ! \\ f_t(\mathbf{u}(a_1; \ldots; a_k); b_1; \ldots; b_m) &= f_t(a_1; \ldots; a_k; b_1; \ldots; b_m) \end{aligned}$$

With the above translation rules, we can now translate finite $\mathrm{PGA_{ul}}$-programs to their $\mathrm{DLTA_f}$-versions. A complete translation would require a translation of repetition as well. This, however, is quite a complex task. The reason for that becomes clear when considering examples like these:

$$\begin{gathered} (a; b; +c)^\omega \\ (+a; +b; +c)^\omega \\ (a; +b; \#5; c; +d;)^\omega \end{gathered}$$

Because of the behavior of $+c$, we get into trouble here if we attempt to use the regular translation. The problem is that $+c$ can possibly skip the first instruction of the next repetition loop, which is behavior that is hard to translate without explicitly introducing this variant of repetition ($^\omega$) in $\mathrm{DLA_f}$. The same problem arises with the jump instruction. At first glance, the best solution there is to introduce the jump instruction to $\mathrm{DLA_f}$ as well. In that case the second canonical form of PGA-programs comes in handy, as it is designed to manipulate expressions with repetition such that no infinite jumps occur.

Since this case study is meant as a relatively clear example of how to use $\mathrm{DLA_f}$ to model side effects in other systems such as PGA, it is beyond our interest here to present these rather complex translations of repetition. Instead, we restrict ourselves to finite $\mathrm{PGA_{ul}}$-programs and leave the relational semantics for $\mathrm{DLA_f}$ which models side effects, as the basis for future work on PGA involving repetition.

8.4 A working example

In this section I will present a working example of the translation from finite $\mathrm{PGA_{ul}}$-programs, which we write as $\mathrm{PGA_{ul}^{fin}}$, to $\mathrm{DLTA_f}$. In addition, I will show that we get sufficiently similar results if we first translate $\mathrm{PGA_{ul}^{fin}}$ to $\mathrm{DLTA_f}$ compared to first projecting $\mathrm{PGA_{ul}^{fin}}$ to $\mathrm{PGA_u^{fin}}$ and then translating that to $\mathrm{DLTA_f}$. To be exact, we are going to show that the following diagram defines a program transformation E on finite deterministic programs in $\mathrm{DLTA_f}$:

$$\begin{array}{ccc} \mathrm{PGA_{ul}^{fin}} & \xrightarrow{f_t} & \mathrm{DLTA_f} \\ {\scriptstyle \mathrm{pgaul2pgau}}\downarrow & & \vdots\, E \\ \mathrm{PGA_u^{fin}} & \xrightarrow[f_t]{} & \mathrm{DLTA_f} \end{array}$$

Here E is a reduction function on $\mathrm{DLTA_f}$ that yields deterministic $\mathrm{DLTA_f}$-programs where occurrences of $\between$ and $\between$ have been eliminated.

For the working example, we return to a variant of our running example. Consider the $\mathrm{PGA_{ul}^{fin}}$-program

$$X = +([x := x+1]\top \between x = 2); \mathbf{u}(w[x = 2]; !); w[x \neq 2]; !$$

where $w[...]$ suggests a write command. This is a program of the form

$$+(b \between c); \mathbf{u}(d; !); e; !$$

with $b = [x := x{+}1]\top, c = (x = 2), d = w[x = 2]$ and $e = w[x \neq 2]$. Thus, we get the following translation, where we for clarity have underlined the instruction

that we are going to translate next:

$$
\begin{aligned}
&f_t(\underline{+(b \mathbin{\diamondsuit\!\!\wedge} c)}; \mathbf{u}(d;!); e;!) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(\underline{\mathbf{u}(d;!)}; e;!)) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(e;!)) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(\underline{d};!; e;!)) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(e;!)) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); d; f_t(\underline{!}; e;!)) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(e;!)) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); d;!) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); f_t(\underline{e};!)) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); d;!) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); e; f_t(\underline{!})) = \\
&(?(b \mathbin{\diamondsuit\!\!\wedge} c); d;!) \cup (?\neg(b \mathbin{\diamondsuit\!\!\wedge} c); e;!)
\end{aligned}
$$

So there we have it: if we replace the shorthands with their original instructions or formulas again, we get the following $\mathrm{DLTA_f}$-program, which we baptize $d\pi_{ul}$:

$$
\begin{aligned}
d\pi_{ul} = \; & (?([x := x+1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2)); w[x = 2];!) \\
& \cup \\
& (?\neg([x := x+1]\top \mathbin{\diamondsuit\!\!\wedge} (x = 2)); w[x \neq 2];!)
\end{aligned}
$$

Clearly, given model M, ${}_g[\![f_t(X)]\!]^M_h$ implies that $h = g[x \mapsto g(x)+1]$. So, if $g(x) = 1$, the instruction $w[x = 2]$ is executed, after which the program terminates, while for $g(x) \neq 1$, the instruction $w[x \neq 2]$ is executed after which the program terminates.

Now let $Y = \mathrm{pgaul2pgau}(X)$, so

$$
Y = \mathbf{u}\big(+ ([x := x+1]\top); \mathbf{u}(+(x = 2); \#2); \#2\big); \mathbf{u}(w[x = 2];!); w[x \neq 2];!
$$

We compute

$$
\begin{aligned}
f_t(Y) = f_t(+([x := x+1]\top); \mathbf{u}(+(x = 2); \#2); \#2; \mathbf{u}(w[x = 2];!); w[x \neq 2];!) \\
= \; & (?([x := x+1]\top); f_t(+(x = 2); \#2; \#2; \mathbf{u}(w[x = 2];!); w[x \neq 2];!)) \\
& \cup \\
& (?\neg([x := x+1]\top); f_t(\#2; \mathbf{u}(w[x = 2];!); w[x \neq 2];!)) \\
= \; & (?([x := x+1]\top); \; (\\
& \qquad (?(x = 2); f_t(\#2; \#2; \mathbf{u}(w[x = 2];!); w[x \neq 2];!)) \\
& \qquad \cup \\
& \qquad (?\neg(x = 2); f_t(\#2; \mathbf{u}(w[x = 2];!); w[x \neq 2];!)) \\
& \qquad) \\
&) \\
& \cup \\
& (?\neg([x := x+1]\top); w[x \neq 2];!) \\
= \; & (?([x := x+1]\top); \; (\\
& \qquad (?(x = 2); w[x = 2];!) \\
& \qquad \cup \\
& \qquad (?\neg(x = 2); w[x \neq 2];!) \\
& \qquad) \\
&) \\
& \cup \\
& (?\neg([x := x+1]\top); w[x \neq 2];!)
\end{aligned}
$$

Note that for each model M and initial valuation g, $M \not\models_g \neg([x := x+1]\top)$, so

$$
\begin{aligned}
{}_g[\![f_t(Y)]\!]^M_h \text{ iff } {}_g[\![?([x := x+1]\top);\ & (\\
& (?(x = 2); w[x = 2]; !)\\
& \cup\\
& (?\neg(x = 2); w[x \neq 2]; !)\\
&)]\!]^M_h
\end{aligned}
$$

Thus, writing $d\pi_u$ for the rightmost deterministic $\mathrm{DLTA_f}$-program, we find

$${}_g[\![f_t(Y)]\!]^M_h \text{ iff } {}_g[\![d\pi_u]\!]^M_h$$

We now need to ask ourselves if $d\pi_u$ is 'sufficiently similar' to the earlier derived $d\pi_{ul}$. Intuitively, we would say that in this working example, this indeed is the case. After all, $[x := x+1]\top$ always yields true, so the truth of $[x := x+1]\top \between (x = 2)$ depends solely on the Boolean reply that $x = 2$ yields. It therefore does not matter if we lift $?[x := x+1]\top$ out of the union, which is essentially what we have done in the case of $d\pi_u$.

We can call two programs 'sufficiently similar' if they evaluate the same single instructions, not being tests, or primitive formulas in the same order. We can formalize that notion with the following proposition:

Proposition 11. *Let X be a program in PGA^{fin}_{ul}, let $d\pi_{ul} = f_t(X)$ and let $d\pi_u = f_t(\mathrm{pgaul2pgau}(X))$. Let model M be given and let g be an initial valuation such that there exists a valuation h such that ${}_g[\![d\pi_{ul}]\!]^M_h$. Then*

$${}_g[\![d\pi_{ul}]\!]^M_h \textit{ iff } {}_g[\![d\pi_u]\!]^M_h$$

and the same single instructions, not being tests, and primitive formulas are evaluated in the same order during evaluation of $d\pi_{ul}$ and $d\pi_u$ given g.

As said, we do not consider repetition as program constructor in our case study. Furthermore, our model of side effects is limited to terminating programs, as opposed to programs that can either end in termination or in divergence. A proof of this proposition might be found, but is for these reasons perhaps not very much to the point. In Chapter 9 (Conclusions) we return to this issue.

It is, however, worthwhile to check the proposition for our working example. Recall that we have the following $d\pi_{ul}$ and $d\pi_u$:

$$
\begin{aligned}
d\pi_{ul} = {} & (?([x := x+1]\top \between (x = 2)); w[x = 2]; !)\\
& \cup\\
& (?\neg([x := x+1]\top \between (x = 2)); w[x \neq 2]; !)\\
d\pi_u = {} & ?([x := x+1]\top);\\
& (?(x = 2); w[x = 2]; !)\\
& \cup\\
& (?\neg(x = 2); w[x \neq 2]; !)
\end{aligned}
$$

It is not hard to check in this case that for any model M and initial valuation g such that $d\pi_{ul}$ can be evaluated, ${}_g[\![d\pi_{ul}]\!]^M_h$ iff ${}_g[\![d\pi_u]\!]^M_h$. It is also easy to see that the same single instructions, not being tests, and primitive formulas are evaluated (in the same order). After all, $d\pi_{ul}$, first evaluates the primitive

formulas $[x := x + 1]\top$ and $x = 2$ and uses those to determine the reply of $[x := x + 1]\top \between (x = 2)$. Depending on the reply, it then either evaluates the single instructions $w[x = 2]$ and !, or $w[x \neq 2]$ and !.

Almost the same goes for $d\pi_u$. It first evaluates the primitive formula $[x := x + 1]\top$ and depending on the reply (which happens to be always true), either stops evaluation (which therefore is never the case) or continues with the evaluation of primitive formula $x = 2$. Depending on the reply, it like $d\pi_{ul}$ then either evaluates the single instructions $w[x = 2]$ and !, or $w[x \neq 2]$ and !. So at least in our working example, Proposition 11 holds.

In a similar way, we can analyze the $\mathrm{PGA}_{\mathrm{ul}}^{\mathrm{fin}}$-program

$$+(\neg[x := x + 1]\top \between x = 2); \mathbf{u}(w[x = 2]; !); w[x \neq 2]; !$$

We can compute $d\pi_{ul} = f_t(X)$:

$$\begin{aligned}
f_t(X) &= f_t(+(\neg[x := x + 1]\top \between x = 2); \mathbf{u}(w[x = 2]; !); w[x \neq 2]; !) \\
&= (?(\neg[x := x + 1]\top \between x = 2); f_t(\mathbf{u}(w[x = 2]; !); w[x \neq 2]; !) \\
&\quad \cup \\
&\quad (?\neg(\neg[x := x + 1]\top \between x = 2); f_t(w[x \neq 2]; !)) \\
&= (?(\neg[x := x + 1]\top \between x = 2); f_t(w[x = 2]; !; w[x \neq 2]; !) \\
&\quad \cup \\
&\quad (?\neg(\neg[x := x + 1]\top \between x = 2); w[x \neq 2]; f_t(!)) \\
&= (?(\neg[x := x + 1]\top \between x = 2); w[x = 2]; f_t(!; w[x \neq 2]; !) \\
&\quad \cup \\
&\quad (?\neg(\neg[x := x + 1]\top \between x = 2); w[x \neq 2]; !) \\
&= (?(\neg[x := x + 1]\top \between x = 2); w[x = 2]; !) \\
&\quad \cup \\
&\quad (?\neg(\neg[x := x + 1]\top \between x = 2); w[x \neq 2]; !)
\end{aligned}$$

We once again define $Y = \mathrm{pgaul2pgau}(X)$, so

$$Y = \mathbf{u}\big(- ([x := x + 1]\top); \#2; +(x = 2)\big); \mathbf{u}(w[x = 2]; !); w[x \neq 2]; !$$

We compute

$$
\begin{aligned}
f_t(Y) &= f_t(-([x := x+1]\top); \#2; +(x=2); \mathbf{u}(w[x=2];!); w[x \neq 2];!) \\
&= (?(\neg([x := x+1]\top)); f_t(\#2; +(x=2); \mathbf{u}(w[x=2];!); w[x \neq 2];!) \\
&\quad \cup \\
&\quad (?\neg(\neg([x := x+1]\top)); f_t(+(x=2); \mathbf{u}(w[x=2];!); w[x \neq 2];!) \\
&= (?(\neg([x := x+1]\top)); f_t(\mathbf{u}(w[x=2];!); w[x \neq 2];!) \\
&\quad \cup \\
&\quad (?\neg(\neg[x := x+1]\top); \ (?(x=2); f_t(w[x=2];!; w[x \neq 2];!)) \\
&\qquad\qquad\qquad\qquad \cup \\
&\qquad\qquad\qquad\qquad (?\neg(x=2); f_t(w[x \neq 2];!)) \\
&\qquad\qquad\qquad\qquad) \\
&= (?(\neg([x := x+1]\top)); w[x=2];!) \\
&\quad \cup \\
&\quad (?\neg(\neg([x := x+1]\top)); \ (?(x=2); w[x=2];!) \\
&\qquad\qquad\qquad\qquad \cup \\
&\qquad\qquad\qquad\qquad (?\neg(x=2); w[x \neq 2];!) \\
&\qquad\qquad\qquad\qquad)
\end{aligned}
$$

We can directly eliminate a situation: $\neg([x := x+1]\top)$ is false for any initial valuation g. Thus, writing $d\pi_u$ for the second part of the topmost union:

$$
\begin{aligned}
d\pi_u = \ ?\neg(\neg([x := x+1]\top)); \ & (\\
& (?(x=2); w[x=2];!) \\
& \cup \\
& (?\neg(x=2); w[x \neq 2];!) \\
&)
\end{aligned}
$$

we get given model M for any initial valuation g

$$ {}_g[\![f(Y)]\!]_h^M \text{ iff } {}_g[\![d\pi_u]\!]_h^M $$

We can check in similar fashion as before that Proposition 11 holds (for any initial valuation g). We can conclude that at least for these working examples, the mentioned proposition is valid. As said, we leave the proof for future work.

This case study started from the abstract approach to attempt decomposition of complex steering fragments in instruction sequences in $\mathrm{PGA}_{\mathrm{ul}}^{\mathrm{fin}}$ as advocated in [5]. We show that we can apply this approach to a rather concrete instance in imperative programming (namely the set A of basic instructions given in this chapter) and we obtain some interesting results. In the first place, it inspired our definition of $\mathrm{DLTA}_{\mathrm{f}}$ and the analysis and classification of side effects as discussed in this thesis. Secondly, by the preservation property formulated in Proposition 11, it justifies our proposal for the projection function pgaul2pgau. It is an interesting result that we are able to show that the projection pgaul2pgau, which does not have to anything to do with valuations, preserves the relational semantics (and therefore side effects) of a program via the diagram at the beginning of this section, which is based on a very natural translation.

9

Conclusions and future work

In this thesis I have given a formal definition of side effects. I have done so by modifying a system for modelling program instructions and program states, Quantified Dynamic Logic, to a system called $\mathrm{DLA_f}$ (Dynamic Logic with Assignments as Formulas), which in contrast to QDL allows assignments in formulas and makes use of short-circuit evaluation. I have shown the underlying logic in those formulas to be a variant of short-circuit logic called repetition-proof short-circuit logic.

Using $\mathrm{DLA_f}$ I have defined the actual and the expected evaluation of a single instruction. The side effects are then defined to be the difference between the two. I have given rules for composing those side effects in single instructions, thus scaling up our definition of side effects to a definition of side effects in deterministic $\mathrm{DLA_f}$-programs. Using this definition I have given a classification of side effects, introducing as most important class that of marginal side effects. Finally, I have shown how to use our system for calculating the side effects in a real system such as PGA.

Our definition gives us an intuitive way to calculate the side effects in a program. Because of the definition in terms of actual and expected evaluation, one can easily adapt the system to ones own needs without having to change the definition of side effects. All one has to do is update the expected evaluation of a single instruction, or if an entirely new single instruction is added to the system, define the actual and expected evaluation for it.

In Chapter 5 we have seen how a sound axiomatization of the formulas in $\mathrm{DLA_f}$ can be given using the signature $\{\top, \bot, \_ \lhd \_ \rhd \_\}$. I have not used this signature in the first place because I wanted to stick to the conventions in dynamic logic. It is noteworthy, however, that this alternative and possibly more elegant signature exists, especially because an axiomatization can be given for it.

The definition of side effects given here can point the way to a lot more research. I can see future work being done in the following areas:

- I do not want to claim that the instructions I have defined in $\mathrm{DLA_f}$ are exhaustive. Finding out what possible other instructions might have to be added to $\mathrm{DLA_f}$ can be an interesting project.
- Another possible subject for future work is the issue of 'negative' side

effects I briefly touched upon in Section 6.5. It is an open question whether or not we should allow situations in which 'negative' side effects occur and if so, how we should handle them.

- In this thesis, we have mostly been looking at imperative programs. It should be interesting to see if our definition can be extended to, for example, functional programs. Perhaps the work done by Van Eijck in [10], in which he defines functional programs making use of program states, can be used for this.

- Another interesting question, which has been raised before in Chapters 2 and 6, is that of side effects in non-deterministic programs. It warrants further research if it is reasonable to talk about side effects there. One can imagine that if the set of side effects in all possibilities of a non-deterministic program are the same, the side effects of the whole can be defined as exactly that set. What needs to be done if that's not the case however, or if we should even want to define side effects of such programs, are open questions.

- In Chapter 7, the concept of marginal side effects was introduced and the suggestion was made that this notion can be linked to claims about how well-written a program is. I have not pursued such claims, but can imagine further research being done in that area.

- To develop a direct modelling of side effects for the variant of PGA discussed in Chapter 8, one can introduce valuation functions as program states and define a relational meaning that separates termination from deadlock/inaction, say

  $$ {}_g[\![X]\!]_h $$

  The idea of this would be to evaluate X as far as possible, which is a reasonable requirement if X is in second canonical form. In addition, we could define a termination predicate, e.g. $\mathrm{Term}(X, g)$, which states that X terminates for initial valuation g. Using this we could define a "behavioral equivalence" on programs X and Y as follows:

  $$ \forall g,_g [\![X]\!]_h \text{ iff } {}_g[\![Y]\!]_h \text{ AND } \mathrm{Term}(X, g) \text{ iff } \mathrm{Term}(Y, g) $$

  Using this, Proposition 11 can probably be proven, especially considering the in Chapter 4 proven property of $\mathrm{DLA_f}$ that any program can be rewritten into a form in which its steering fragments only contain primitive formulas and their negations.

- Also mentioned in Chapter 8 is the possibility to introduce extra logical operators, namely Logical Left And (LLAnd) and its dual Logical Left Or (LLOr). Introducing these in $\mathrm{DLA_f}$ is fairly straight-forward: one only needs to define its truth in M:

  $$ M \models_g \phi_1 \mid \phi_2 \text{ iff } M \models_g \phi_1 \,{}^{\circ}\!\vee\, \phi_2 \qquad \text{(DLA7c)} $$
  $$ M \models_g \phi_1 \,\&\, \phi_2 \text{ iff } M \models_g \phi_1 \,{}_{\circ}\!\wedge\, \phi_2 \qquad \text{(DLA7d)} $$

  as well as update the program extraction function:

  $$ \Pi_g^M(\phi_1 \Box \phi_2) = \Pi_g^M(\phi_1); \Pi_h^M(\phi_2) \text{ if } {}_g[\![\Pi_g^M(\phi_1)]\!]_h^M \text{ and } \Box \in \{|, \&\} $$

To introduce the same operator in $\mathrm{PGA}_{\mathrm{ul}}$, projection functions in the same style as the ones given in Chapter 8 for SCLAnd and SCLOr need to be defined.

- Another possible matter for further study is whether side effects can be used in natural language. In the Introduction, we have already seen that they can occur in the pregnant wife example, where your wife told you to do the grocery shopping if she did not call you, which she later did, but to tell you that she was pregnant. Possibly there is a role for side effects when explaining misunderstandings. There is no doubt that side effects can be the cause of misunderstandings. The pregnant wife example illustrates that: you could decide to do grocery shopping to be on the safe side after her call, claiming her call indicated you might have to shop, only to run into your wife at the store also shopping (who, of course, didn't want to convey the message that you should shop at all).

  When we take the Dynamic Epistemic Logic system mentioned in [12], the knowledge of two communicating agents is captured by an epistemic state, one for each agent. The agents also have an epistemic state for what they think is the (relevant) knowledge of the other agent with whom they are in conversation. A misunderstanding has occurred when an agent updates his own epistemic state in a different way than the other agents expects him to. There are a lot of ways in which this can happen, but relevant for us is that one of those ways is, when a side effect from an utterance occurs of which one of the agents is not aware.

  If one of the agents is aware of the side effect and also of the fact the other agent might not be aware of it, it may be recommended to point out this side effect to the other agent. In our example of the pregnant wife calling, this would mean that you would have to ask your wife on the phone that the fact she called leaves you in doubt about the grocery shopping. Naturally, though, we recommend a more enthusiastic response to the news she is pregnant first.

Bibliography

[1] J.A. Bergstra. Steering Fragments of Instruction Sequences. arXiv:1010.2850, October 2010.

[2] J.A. Bergstra, J. Heering and P. Klint. Module algebra. In: *Journal of the ACM*, Volume 37, Number 2, pp. 335-372, 1990.

[3] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. In: *Journal of Logic and Algebraic Programming*, volume 51, pp. 125-156, 2002.

[4] J.A. Bergstra and A. Ponse. Proposition algebra. In: *ACM Transactions on Computational Logic*, Volume 12, Number 3, Article 21, 2011.

[5] J.A. Bergstra and A. Ponse. Short-Circuit Logic. arXiv:1010.3674, 2011.

[6] H. Böhm. Side effects and aliasing can have simple axiomatic descriptions. In: *ACM Transactions on Programming Language and Systems*, volume 7, number 4, pp. 637-655, 1985.

[7] P.E. Black and P.J. Windley. Inference rules for programming languages with side effects in expressions. In: J. von Wright, J. Grundy and J. Harrison (eds.), *Theorem Proving in Higher Order Logics: 9th International Conference*, pp. 51-60. Springer-Verlag, Berlin, Germany, 1996.

[8] P.E. Black and P.J. Windley. Formal Verification of Secure Programs in the Presence of Side Effects. `http://phil.windley.org/papers/hicss31.ps`, 1998.

[9] P. Dekker. A Guide to Dynamic Semantics. `http://www.illc.uva.nl/Publications/ResearchReports/PP-2008-42.text.pdf`, 2008.

[10] J. van Eijck, Purely Functional Algorithm Specification. `http://homepages.cwi.nl/~jve/pfas/`, 2011.

[11] J. van Eijck and M. Stokhof. The Gamut of Dynamic Logics. In: D. Gabbay and J. Woods (eds.), *Handbook of the History of Logic*, volume 7, pp. 499-600. Elsevier, 2006.

[12] J. van Eijck and A. Visser. Dynamic Semantics. In: E. Zalta (ed.), *Stanford Encyclopedia of Philosophy*, Fall 2010 Edition. 2010.

[13] R. Goldblatt. Axiomatising the Logic of Computer Programming. Springer-Verlag, Berlin and New York, 1982.

[14] D. Harel. First-Order Dynamic Logic. Number 68 of *Lecture Notes in Computer Science.* Springer, Berlin, 1979.

[15] D. Harel. Dynamic Logic. In: D. Gabbay and F. Günthner (eds.), *Handbook of Philosophical Logic*, Volume II, pp. 497-604, 1984.

[16] C.A.R. Hoare. A couple of novelties in the propositional calculus. In: *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 31(2), pp. 173-178, 1985.

[17] M. Norrish. An abstract dynamic semantics for C. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-421.pdf`, Computer Laboratory, University of Cambridge, Technical Report, 1997.

[18] A. Ponse. Program algebra with unit instruction operators. In: *Journal of Logic and Algebraic Programming*, volume 51, pp. 157-174, 2002.

[19] V. Pratt. Semantical considerations on Floyd-Hoare logic. In: P. Abrahams, R. Lipton and S. Bourne (eds.), *Proceedings 17th IEEE Symposium on Foundations of Computer Science*, pp. 109-121. IEEE Computer Science Society Press, Long Beach, CA, 1976.

[20] V. Pratt. Application of modal logic to programming. *Studia Logica*, Volume 39, pp. 257-274, 1980.

[21] A. van Wijngaarden et al. Revised report on the algorithmic language Algol 68. In: *Acta Informatica*, Volume 5, Numbers 1-3, pp. 1-236, 1975.