# Hypernet Semantics of Programming Languages

by

Koko Muroya

A thesis submitted to the University of Birmingham
for the degree of DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
June 2019

# Abstract

Comparison is common practice in programming, even regarding a single programming language. One would ask if two programs behave the same, if one program runs faster than another, or if one run-time system produces the outcome of a program faster than another system. To answer these questions, it is essential to have a formal specification of program execution, with measures such as result and resource usage.

This thesis proposes a semantical framework based on abstract machines that enables analysis of program execution cost and direct proof of program equivalence. These abstract machines are inspired by Girard's Geometry of Interaction, and model program execution as dynamic rewriting of graph representation of a program, guided and controlled by a dedicated object (*token*) of the graph. The graph representation yields fine control over resource usage, and moreover, the concept of *locality* in analysing program execution. As a result, this framework enjoys novel flexibility, with which various evaluation strategies and language features, whether they are effects or not, can be modelled and analysed in a uniform way.

# Acknowledgements

Throughout three years of studying in Birmingham I had far more opportunities and experiences than I could have ever imagined, both inside and outside academia. I owe many of these to my supervisor and friend Dan R. Ghica. I learned a lot from interactions with him, and he deserves my special thanks for his continuous support and positivity.

Thanks to Alexandra Silva for the invitation to the Bellairs workshop in March 2018, where the work on Spartan got going, to Nando Sola for giving me a unique opportunity to present my work to functional programmers in Lambda World Cádiz, and to David R. Sherratt in Bath for his interest and hospitality.

I am grateful to the members of the ever-growing Birmingham theory group, especially my thesis group members Paul Levy and Uday Reddy, for giving me the sense of community. Many thanks go to Steven W. T. Cheung and Todd Waugh Ambridge who have been my good colleagues, collaborators and friends. I am also grateful to my external examiner, Ugo Dal Lago, for his encouraging comments.

Meeting people has given me motivation during my Ph.D. study. I would especially like to mention Ahmed Al-Ajeli, Dong Li, Yuning Feng and Simon Fong who enriched my life. Thanks to my friends and family for their help and support, sometimes all the way from Japan, without which this thesis could have never been completed.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Context and motivation

### 1.1.1 Abstract machines for programming languages

Programming languages are for humans to communicate with computers. In this one-way communication, humans write programs, and computers execute programs. While the execution of programs is *implemented* by compilers and interpreters, it is *specified* by semantic models. The specification can be in many terms, such as the outcome, process, and properties, of program execution.

Given a computer program, executing it using one implementation is typically not the only concern. Comparison between programs, or between implementations, comes in various forms. One could ask if two programs have the same output, or if one program has better execution cost than the other. It is desirable to have the same output of a single program regardless of the choice of implementation, so that the program is portable. The choice would depend on other measures, such as execution cost, that can vary between implementations.

The comparison can only be possible if measures like output and execution cost are formalised, independently of implementation details. What we need is a formal specification of program execution that works across programs and implementations.

*Abstract machines* are the oldest of such formal specification [Landin, 1964]. They model program execution by changing a *configuration* step-by-step. Each configuration typically consists of a program fragment (*code*) under evaluation, and some data structures, such as stacks and lookup tables, that record necessary information to determine a possible change of the configuration. In *loc. cit.*, the first abstract machine for the lambda-calculus, called the SECD machine, was introduced. The machine is named after the four components of the configurations it uses: Stack, Environment, Control and Dump. While the control represents the code under evaluation, the other three components serve as the auxiliary data structures. The stack records intermediate results, the environment serves as a lookup table for variables, and the dump records computations that are to be resumed once the current code is evaluated.

Using an abstract machine, the whole process of program execution can be obtained as a sequence of configurations, or a history of configuration changes. This sequence enables us to analyse both the result and the cost of execution. The execution result is represented by the final configuration of the sequence, and the space cost can be measured using the size of each configuration. The time cost can be estimated by adding up time usage of configuration changes. If all changes finish in constant time, the time cost can simply be measured in terms of the number of changes. If not, which is often the case, time usage of each configuration change needs to be estimated. This is possible by examining how much each component, namely data structure, of the configuration is modified each time.

Abstract machines are deemed "abstract", and do not immediately yield implementations of program execution such as compilers, because they abstract away details of computer architectures. At the same time, they are not as abstract as other formal specifications, such as denotational semantics and operational semantics, because they do not abstract away data structures and maintain them explicitly. This balance lets abstract machines serve as a basis of implementations of program

execution, such as compilers, and provide a cost measure of program execution that is independent of implementation details.

Abstract machines may be abstract enough to specify execution steps and analyse execution cost, but they are considered to be too concrete to reason about programs, as recognised by Plotkin [1981, 2004] in his development of *structural operational semantics*. Programs are structured according to a formal grammar (*syntax*), and the syntactical, structural, information about programs is vital in reasoning. For example, even the simple notion of program fragment, or sub-program, is syntactical. Comparison between two programs would therefore start with recognising their syntactical difference, which is given in terms of sub-programs. Nevertheless, the structural information is obscured by the use of concrete data structures in abstract machines. Each data structure captures an aspect of the structural information, if any, and each modification of a configuration typically concerns only a part of the configuration. It is hard to recover the structural information from a sequence of configuration changes, which represents the process of program execution.

### 1.1.2 Comparison between implementations

There are two main measures of program execution: result and cost. The execution result is commonly given by a single value that a program returns, but it can be enriched in the presence of certain language features. Interactive I/O equips the return value with an entire history of interaction between the program and users, non-determinism yields multiple possible return values, and probabilistic features yield a probability distribution of return values. In any case, the result of a single program is supposed to be the same across implementations, which makes the program portable.

On the other hand, execution cost can vary between implementations, leaving a certain freedom in managing efficiency. One may prefer better space efficiency, or better time efficiency, and it is well known that one can be traded off for the other.

For example, time efficiency can be improved by caching more intermediate results, which increases space cost, whereas bounding space requires repeating computations, which adds to the time cost.

The execution cost is a key factor in comparison between implementations of a programming language. Each implementation should have its own specification of execution cost. Yet we need a way to compare various specifications. To make this possible, a language should come with a cost measure that is applicable to all implementations. The measure should be concrete enough to be realistic, but at the same time abstract enough to be independent of implementation details. Implementations can then be specified and compared uniformly by means of the cost measure.

Although an abstract machine can serve as a basis of implementations and provide a cost measure, its measure cannot necessarily be used for all possible implementations of the language. The main reason is that there is no generic abstract machine that models all possible implementations. There is no canonical way to design an abstract machine for a given programming language, and each abstract machine has its own design choices, and accordingly, its own scope of implementations. Comparison between implementations can be alternatively done by comparing abstract machines, but these machines come with their own cost measures.

Recent studies by Accattoli and Dal Lago [2016], Accattoli et al. [2014], Accattoli [2017] establish a methodology with which one can compare and classify abstract machines in terms of their execution cost. They propose a cost measure that is with respect to *evaluation strategies*, not to programming languages. Evaluation strategies can be used to characterise programming languages, but different strategies could yield the same execution result, which means that the characterisation may not be unique.

In the setting of functional programming, evaluation strategies determine how an argument of a function is evaluated, and often imply how intermediate results

are copied, discarded, cached or reused, which affects execution cost. Examples include call-by-name, call-by-need (or, lazy), and call-by-value evaluation strategies. In the call-by-name evaluation strategy, evaluation of a function body proceeds until its arguments are actually required. Each argument gets evaluated as many times as it is requested. In the call-by-need, or lazy, strategy, computation proceeds in the same way, but each argument is evaluated at most once. Once an argument is requested, its evaluation result is cached for later use, preventing re-evaluation of the same argument. The call-by-value strategy evaluates all arguments first and caches the results, before evaluating the function body. This means that each argument is evaluated exactly once. The call-by-value strategy can be further classified in terms of the ordering of argument evaluations.

If evaluation of some of the arguments does not terminate, the call-by-value strategy does not terminate, whereas the other two strategies might terminate. These two strategies could terminate namely when these non-terminating arguments are not used. If some of the arguments do not have a unique evaluation result, the call-by-name and call-by-need strategies could have different overall results. However, in programming languages where evaluation always terminates with a unique result, the choice of strategy should only affect the overall cost, but not the overall result.

### 1.1.3 Comparison between programs

Orthogonal to the comparison between implementations is comparison between programs. It also requires specification of program execution and formalisation of a measure of execution, such as measures of result and cost. Programs, or more generally program fragments, can be compared with respect to one measure or combination of multiple measures.

One classical question is comparison with respect to execution result, asking whether two program fragments have the same behaviour. The standard formalisation of this question is given as *observational equivalence* [Morris Jr, 1969]. Two

whole programs are observationally equivalent if their execution yields the same output. Program fragments are observationally equivalent if any two whole programs whose only difference is these fragments are observationally equivalent.

Another established question, formalised as *improvement* [Moran and Sands, 1999], is whether one program simulates another program with less resource usage. This comparison concerns both execution result and cost, and it can be generalised to program fragments in the same way as observational equivalence.

These questions of observational equivalence and improvement are of interest to programmers as well as language developers. When programmers update a software, they can make sure there is no *regression*, i.e. bugs introduced by the update, by proving that the old and new versions of the software are observationally equivalent. Observational equivalence can also be used to verify distributed algorithms with respect to their sequential specification. Compilers typically employ series of program transformations, to turn a source code into an assembly code. Correctness of these transformations can be verified using observational equivalence. Moreover, some transformations are intended to optimise a given program, which can be validated using improvement.

To prove observational equivalence between program fragments, one needs to inspect all possible *contexts*, i.e. whole programs without the fragments, and their interaction with the fragments throughout execution. As proof methods of observational equivalence, *logical relations* [Plotkin, 1973, Statman, 1985] enable us to analyse the interaction in terms of types, and *applicative bisimulations* [Abramsky, 1990] identify function application as the fundamental interaction between contexts and fragments.

Alternatively, a specification of program execution itself can be designed by formalising interaction between program components. *Game semantics* [Abramsky et al., 2000, Hyland and Ong, 2000] is one example, where programs are interpreted in terms of possible interactions they can have. It solved the full abstraction prob-

lem, which is of giving an equivalent interpretation to program fragments if and only if these fragments are observationally equivalent, firstly for the functional programming language PCF [Plotkin, 1977].

The more features a programming language has, the more problematic it becomes to inspect all possible interactions between contexts and fragments. Extra language features could enable contexts to distinguish more program fragments, and hence to break some existing observational equivalences.

For example, mutable state enables contexts to distinguish some syntactically identical program fragments. This leads to violation of certain standard observational equivalence, such as the identity law $M$ `=` $M \simeq$ `true` of the equality operation (‘`=`’), where $M$ is an arbitrary program fragment. In the following two programs, written in the style of OCaml, a context exploits mutable state and indeed violates an instance of the identity law: (`f ()`) `=` (`f ()`) $\simeq$ `true`, where `f` is a function defined elsewhere and is applied to the unit value `()`. The context distinguishes the two syntactically identical arguments `f ()` of the equality operation.

```
1 ;; a program that returns false
2 let b = ref true in
3 let f _ =
4   b := not !b;
5   !b
6 in
7 (f ()) = (f ())
```

```
1 ;; a program that returns true
2 let b = ref true in
3 let f _ =
4   b := not !b;
5   !b
6 in
7 true
```

The distinguishing context appears in lines 2–6 of each program. It creates a mutable boolean state $b$, and provides a definition of the function $f$. The mutable state can be accessed through an operation ‘`:=`’ for updating a stored value, and an operation ‘`!`’ for reading a stored value. Upon each call, whatever the argument is, the function

$f$ internally flips the value of the mutable state $b$ (line 4), and returns the flipped value (line 5). The context distinguishes the two function calls ($f$ `()`), of which the first results in `false` and the second results in `true`, and therefore, violates the identity law.

Introduction of new language features could bring extra distinguishing power to contexts, and therefore, it often requires a reformulation of a formal specification to model the distinguishing power as well as the features themselves. The reformulation of a specification further invalidates a proof technique of observational equivalence. This can be observed in the literature about functional programs in the presence of effects, such as state and control.

Since game semantics [Abramsky et al., 2000, Hyland and Ong, 2000] solved the full abstraction problem for the functional programming language PCF, it was adapted to accommodate ground state [Abramsky and McCusker, 1998], control [Laird, 1997], and general references [Abramsky et al., 1998]. While ground state only allows data, such as natural numbers, to be stored, general references (also called higher-order state) has no restriction as to what can be stored.

For logical relations [Plotkin, 1973, Statman, 1985], which is a type-based inductive proof method for observational equivalence, higher-order state poses a challenge by introducing types that are not inductive. To deal with non-inductive types, namely recursive and quantified types, Ahmed [2006] equipped logical relations with *step indices* [Appel and McAllester, 2001]. Step-indexed logical relations were then used to model higher-order state together with abstract types [Ahmed et al., 2009], and to model higher-order state as well as control [Dreyer et al., 2012].

Deviating from applicative bisimulations [Abramsky, 1990], *environmental bisimulations* have been developed to deal with more distinguishing power of contexts, for instance caused by abstract types and recursive types [Sumii and Pierce, 2007], and higher-order state [Sangiorgi et al., 2007, Koutavas and Wand, 2006]. While the deviation is analysed and justified in the presence of effects including state and poly-

morphism [Koutavas et al., 2011], yet another variant of environmental bisimulation was proposed to model control [Yachi and Sumii, 2016].

The aforementioned work [Abramsky and McCusker, 1998, Laird, 1997, Abramsky et al., 1998] on game semantics resulted in so-called Abramsky's cube, a semantical characterisation of combinations of state and control. Abramsky's cube is also studied in terms of logical relations by Dreyer et al. [2012].

Since some common language features, such as mutable state, can indeed violate as basic observational equivalences as the identity law $M = M \simeq \texttt{true}$, it is a natural and important question to ask which observational equivalences are respected or violated by which language features. To answer this question, it would not be enough to analyse the impact that language features have on semantical specifications and their proof methodology of observational equivalence, as in the literature. Language features should rather be analysed in terms of their effect directly on observational equivalences, and this analysis would require a uniform framework that can model all language features. The desirable framework should provide not only a specification of program execution, but also a non-fragile reasoning principle that works in the presence and absence of various language features in a uniform way.

Recall that the notion of improvement formalises comparison between program fragments with respect to both execution result and cost. Dealing with improvement instead of observational equivalence in the presence of rich language features would be yet another complication. The uniform semantical framework, which accommodates all the language features of interest, is still desirable, but the framework should additionally be equipped with a measure of execution cost. Furthermore, the framework should accommodate language features as first-class inhabitants. It is not enough for the framework to provide a set of primitives with which language features can be encoded, because encoding does not necessarily reflect efficiency of the language features.

The common motivation of research regarding improvement seems to be vali-

dation of program transformations used for a lazy programming language such as Haskell. Recent studies with this motivation by Hackett and Hutton [2014], Schmidt-Schauß and Sabel [2017] consider a core lazy language where effects such as state are not first-class inhabitants but could be encoded. For richer languages, known semantical frameworks for observational equivalence have been adapted. Hackett and Hutton [2018] adapt a logical relation for parametric polymorphism to prove improvement. Ghica [2005] proposes game semantics for a concurrent language that can take a given cost measure into account, solving the full abstraction problem with respect to improvement.

## 1.2 Contribution and methodology

### 1.2.1 Two graph-rewriting abstract machines for reasoning

This thesis develops two abstract machines: the *Dynamic Geometry of Interaction Machine* (DGoIM), and the *Universal Abstract Machine* (UAM). Both abstract machines perform strategical graph rewriting, with the latter revising the former. Unlike conventional abstract machines, whose configurations are code accompanied by data structures, these abstract machines work on graph representation of whole programs. Translating inductively-structured programs into graphs, as low-level representation, enables fine control over resources and introduces the novel concept of *locality* in program execution.

The DGoIM is a first step towards an abstract machine, as a semantical framework, with which different specifications of time and space cost can be given in a uniform way. Strategical graph rewriting has flexibility in terms of rewrite rules and strategies of triggering the rewrite rules. This flexibility would enable the DGoIM to explore the design space of abstract machines, and hence make a cost measure of the DGoIM a generic measure of a given programming language.

As a case study, one setting of the DGoIM is proved to efficiently model the

lambda-calculus with call-by-need and call-by-value evaluation strategies. The efficiency is proved using a taxonomy of abstract machines given by Accattoli [2017].

Based on the DGoIM, the UAM aims at a uniform semantical framework with which language features can be analysed in terms of observational equivalences, and possibly improvements, they respect or violate. The UAM specifies program execution of a language called Spartan—a core language whose only primitives are variable binding, name binding and thunking. Everything else, even lambda-abstraction and function application, are extrinsic in Spartan.

In this way, the UAM enables analysis of not just lambda-abstraction and function application, as the DGoIM does, but also other language features. The UAM is dubbed "universal" in the sense of universal algebras; being parametrised by extrinsic operations and their behaviour, the UAM serves as a specification of various languages without the need to change its intrinsic machinery.

In this framework, language features can be accommodated in two different ways: as native operations, and as encoding in terms of other native operations. This makes the UAM potentially suitable for reasoning about improvement, although observational equivalence is the primary concern here.

The UAM is arguably the first known abstract machine that enables direct proofs of observational equivalence. A reasoning principle used in these direct proofs exploits a concept of *locality* that arises in strategical graph rewriting, and the principle is formalised as a *characterisation theorem* (Thm. 4.3.14).

The sequel describes methodological background of these developments, namely strategical graph rewriting and the concept of locality.

### 1.2.2 Token-passing GoI

Geometry of Interaction (GoI) [Girard, 1989], a semantics of linear logic proofs, provides a starting point towards a framework for studying the trade-off between time and space efficiency. The token-passing style of GoI, in particular, gives abstract ma-

chines for the lambda-calculus, pioneered by Danos and Regnier [1996] and Mackie [1995]. These machines evaluate a term of the lambda-calculus by translating the term to a graph, a network of simple transducers, which executes by passing a data-carrying token around.

Token-passing GoI decomposes higher-order computation into local token actions, or low-level interactions of simple components. It can give innovative implementation techniques for functional programs, such as *Geometry of Implementation* compiler [Mackie, 1995], *Geometry of Synthesis* (GoS) high-level synthesis tool [Ghica, 2007], and resource-aware program transformation to a low-level language [Schöpp, 2014a]. The interaction-based approach is also convenient for the complexity analysis of programs, e.g. IntML type system of logarithmic-space evaluation [Dal Lago and Schöpp, 2016], and linear dependent type system of polynomial-time evaluation [Dal Lago and Gaboardi, 2011, Dal Lago and Petit, 2012].

Constant-space execution is essential for GoS, since in the case of digital circuits the memory footprint of the program must be known at compile-time, and fixed. Using a restricted version of the call-by-name language Idealised Algol [Ghica and Smith, 2011] not only the graph, but also the token itself can be given a fixed size. Surprisingly, this technique also allows the compilation of recursive programs [Ghica et al., 2011]. The GoS compiler shows both the usefulness of the GoI as a guideline for unconventional compilation and the natural affinity between its space-efficient abstract machine and call-by-name evaluation. The practical considerations match the prior theoretical understanding of this connection [Danos and Regnier, 1996].

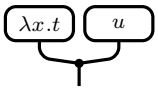
The token passed around a graph simulates graph rewriting without actual rewriting, which is in fact an extremal instance of the trade-off mentioned above. Token-passing GoI keeps the underlying graph fixed and uses the data stored in the token to route it. It therefore favours space efficiency at the cost of time efficiency. The same computation is repeated when, instead, intermediate results could have been cached by saving copies of certain sub-graphs representing values.

### 1.2.3 Interleaving GoI token passing with graph rewriting

The original intention is to lift the token-passing GoI to a framework to analyse the trade-off between time and space efficiency. This can be done by strategically interleaving the GoI token passing with graph rewriting, resulting in an abstract machine. The machine, called the *Dynamic GoI Machine* (DGoIM), is defined as a state transition system with transitions for token passing as well as transitions for graph rewriting. The token holds control over graph rewriting, by visiting redexes and triggering the rewrite transitions.

Graph rewriting offers fine control over caching and sharing intermediate results. Through graph rewriting, the DGoIM can reduce sub-graphs visited by the token, avoiding repeated token actions and improving time efficiency. However, fetching cached results can increase the size of the graph. In short, introduction of graph rewriting sacrifices space while favouring time efficiency. The flexibility given by a fine-grained control over interleaving would enable a careful balance between space and time efficiency.

As a first step in exploration of the flexibility of this machine, we consider the two extremal cases of interleaving. The first extremal case is *passes-only*, in which the DGoIM never triggers graph rewriting, yielding an ordinary token-passing abstract machine. As a typical example, the $\lambda$-term $(\lambda x.t)\, u$ is evaluated like this:




1. A token enters the graph on the left at the bottom open edge.
2. A token visits and goes through the left sub-graph $\lambda x.t$.
3. Whenever a token detects an occurrence of the variable $x$ in $t$, it traverses the right sub-graph $u$, then returns carrying information about the resulting value of $u$.
4. A token finally exits the graph at the bottom open edge.

Step 3 is repeated whenever the argument $u$ needs to be re-evaluated. This passes-only strategy of interleaving corresponds to call-by-name evaluation.

The other extreme is *rewrites-first*, in which the DGoIM interleaves token passing with as much, and as early, graph rewriting as possible, guided by the token. This corresponds to both call-by-value and call-by-need evaluations, with different trajectories of the token. In the case of left-to-right call-by-value, the token enters the graph from the bottom, traverses the left-hand-side sub-graph, which happens to be already a value, then visits the sub-graph $u$ even before the bound variable $x$ is used in a call. The token causes rewrites while traversing the sub-graph $u$, and when it exits, it leaves behind a graph corresponding to a value $v$ such that $u$ reduces to $v$. For right-to-left call-by-value, the token visits the sub-graph $u$ straightaway after entering the whole graph, reduces the sub-graph $u$, to the graph of the value $v$, and visits the left-hand-side sub-graph. Finally, in call-by-need, the token visits and reduces the sub-graph $u$ *only when* the variable $x$ is encountered in $\lambda x.t$.

In this framework, all these three evaluations involve similar tactics for caching intermediate results. Their only difference, which is the timing of cache creation, is realised by different trajectories of the token. Cached values are fetched in the same way: namely, whenever repeated evaluation is required, the sub-graph corresponding to the cashed value is copied. One copy can be further rewritten, if needed, while the original is kept for later reference.

### 1.2.4 Rewrites-first interleaving and locality

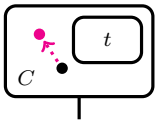
As for a semantical framework to analyse language features in terms of observational equivalence, the main challenge is to have a reasoning principle of inspecting possible contexts. Direct inspection of contexts, as well as their interaction with a particular program fragment throughout execution, is particularly hard with abstract machines.

It turns out, however, that employing the rewrites-first interleaving of the DGoIM

makes the direct inspection possible. Due to the control the token holds over graph rewriting, program execution can be described *locally* in terms of the token and its neighbourhood. The inspection of contexts boils down to local inspection around the token that navigates through contexts: namely, inspection of token's position, data, and rewrites that are triggered by and around the token.

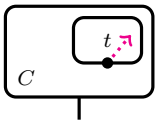
This local inspection can further be applied to analyse interaction that a context $C$ has with a fragment $t$, during execution of the composite program $C[t]$. At the beginning of the execution, the token enters the graph that represents $C[t]$. The graph can be split into two parts, one corresponding to the context $C$ and the other corresponding to the fragment $t$. As the token navigates through the graph $C[t]$, the token position will therefore be either inside $C$, inside $t$ or on the border between the two sub-graphs. By inspecting the token data and rewrites to be triggered, possible scenarios can be classified into the following three:

**Case I: move inside the context.**



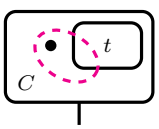

The token (•) moves within the sub-graph $C$, as indicated by magenta on the left. The sub-graph $t$ is not involved in, and hence is irrelevant to, the move.

**Case II: visit to the fragment.**




The token (•) enters the sub-graph $t$. It will navigate through the sub-graph $t$, as indicated by magenta on the left, and may trigger some rewrite. The rewrite possibly involves a part of the sub-graph $C$.

**Case III: rewrite.**



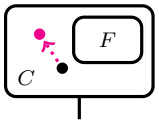

The token (•) is in the sub-graph $C$ and triggers a rewrite. The rewrite may involve a part of the sub-graph $t$, as indicated by magenta on the left.

In this way, the rewrites-first interleaving of the DGoIM enables us to inspect interaction between a context $C$ and a fragment $t$ in an elementary case-by-case

manner, according to how the fragment $t$ is involved in token moves and triggered rewrites that are possible on the composite graph $C[t]$. This leads to a direct, case-by-case, reasoning principle to prove observational equivalence, namely to prove that two program fragments $t$ and $u$ interact with any context $C$ in the same manner. Intuitively, the way the fragment $t$ is involved in token moves and triggered rewrites on the graph $C[t]$ should *coincide* with the way the fragment $u$ is involved in token moves and triggered rewrites on the graph $C[u]$.

Sufficient conditions of observational equivalence can be identified by examining scenarios of this coincidence, using the case analysis based on the local inspection. Example scenarios of the coincidence are as follows:

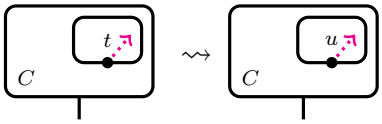
*Any scenario in Case I: move inside the common context.*




($F$ denotes $t$ or $u$.)

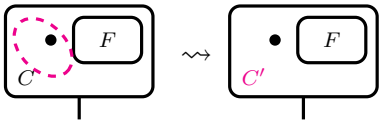
The token moves within the common context $C$, which is regardless of the fragments. The coincidence is in fact always guaranteed in Case I.

*A scenario in Case II: visit to the fragments.*




Whenever the token visits the fragment $t$, triggered rewrites change the fragment $t$ to the other fragment $u$, and hence the visit yields the same result as visiting the fragment $u$. This typically happens when the fragments $t$ and $u$ are taken from reduction, e.g. $t \equiv 1 + 2$ and $u \equiv 3$.

*A scenario in Case III: rewrite.*




($F$ denotes $t$ or $u$.)

The token in the common context $C$ triggers a rewrite that only affect a part of the context $C$. The rewrite hence preserves the fragments.

The last scenario is of particular importance, giving rise to a new concept of *robustness*. It characterises *safe* involvement of the fragments in rewrites, namely where the fragments are respected in the same manner by the rewrites, and provides

a sufficient condition of observational equivalence. Measuring robustness of the fragments reveals when, namely with which rewrites allowed, the fragments can be observationally equivalent.

The main technical result in Chap. 4, a *characterisation theorem* (Thm. 4.3.14), identifies sufficient conditions of observational equivalence in the way described above, by formalising the case analysis based on the local inspection. The theorem provides a way to analyse which observational equivalences are respected by which language features, by identifying robustness as a key sufficient condition. Robustness of program fragments are relative to rewrites, but it can be seen as being relative to language features as well, because behaviour of language features are modelled by rewrites. Therefore, by measuring robustness of program fragments, one can examine which language features can enable the fragments to be observationally equivalent.

Finally, it is worth noting that the concept of locality here is not to be confused with memory locality. Memory locality concerns memory access during program execution. On the other hand, locality, together with robustness, captures the impact that operations in a program make on other parts of the program during execution. Locality could be seen as a generalisation of memory locality, because memory access could be modelled as extrinsic features of Spartan and the UAM.

## 1.3 Thesis outline

Chap. 2 presents the DGoIM with the rewrites-first strategy, as efficient specification for the lambda-calculus with various evaluation strategies. Materials in this chapter have been produced under the supervision of Dan R. Ghica, and presented in jointly-authored papers [Muroya and Ghica, 2017, 2018a], which are currently under consideration as a journal article [Muroya and Ghica, 2018b]. We appreciate encouraging and insightful comments by Ugo Dal Lago and anonymous reviewers

on earlier versions of this work, and thank Steven W. T. Cheung for helping us implement an on-line visualiser.

Chap. 3 presents the language SPARTAN and the UAM, and Chap. 4 presents the local reasoning principle the UAM offers. Materials in these chapters have been produced under the supervision of Dan R. Ghica. Hypernet diagrams were produced by Ghica, and an associated on-line visualiser was implemented by Todd Waugh Ambridge. A preliminary idea of local reasoning was formulated for the lambda-calculus in Waugh Ambridge's MSci thesis, under the joint supervision of Ghica and the author, and was presented at the workshop on Syntax and Semantics of Low-Level Languages (Oxford, 2018).

Chap. 5 discusses some topics that are relevant to both machines DGoIM and UAM. Sec. 5.1 compares the two machines, and Sec. 5.2 gives an overview of related work that adapts the DGoIM to model an unconventional programming paradigm. This work can be seen as a case study of token-guided graph rewriting, which in fact inspired the UAM. Materials mentioned in this section are based on joint work [Cheung et al., 2018, Muroya et al., 2018] with Steven W. T. Cheung, Victor Darvariu, Dan R. Ghica and Reuben N. S. Rowe.

Chap 6 concludes this thesis with discussions about related and future work.

# Chapter 2

# Efficient implementation of evaluation strategies

## 2.1 Outline

This chapter presents a token-guided graph-rewriting abstract machine for call-by-need, left-to-right call-by-value, and right-to-left call-by-value evaluations. We give the abstract machine as the *Dynamic Geometry of Interaction Machine* (DGoIM) with the rewrite-first strategy, which turns out to be as natural as the passes-only strategy for call-by-name evaluation. It switches the evaluations, by simply having different nodes that correspond to the three different evaluations, rather than modifying the behaviour of a single node to suite different evaluation demands. This is a first step in exploration of the flexibility of the DGoIM, which is achieved through controlled interleaving of rewriting and token-passing, and through changing graph representations of terms.

We prove the soundness and completeness of the graph-rewriting machine with respect to the three evaluations separately, using a *sub-machine* semantics, where the word "sub" indicates both a focus on substitution and its status as an intermediate representation. The sub-machine semantics is based on *token-passing* semantics of Sinot [2005, 2006] that makes explicit the two main tasks of abstract machines:

searching redexes and substituting variables.

The time-cost analysis classifies the machine as *efficient* in a taxonomy of abstract machines of Accattoli [2017]. We follow a general methodology developed by Accattoli et al. [2014], Accattoli [2017] for quantitative analysis of abstract machines, however the method cannot be used "off the shelf". Our machine is a more refined transition system with more transition steps, and therefore does not satisfy one of their assumptions [Accattoli, 2017, Sec. 3], which requires one-to-one correspondence of transition steps. We overcome this technical difficulty by building a weak simulation of the sub-machine semantics, which is also used in the proof of soundness and completeness. The sub-machine semantics resembles the storeless abstract machine of Danvy and Zerny [2013], to which the general recipe of cost analysis does apply.

Finally, an on-line visualiser[1] is implemented, in which our machine can be executed on arbitrary closed (untyped) lambda-terms. The visualiser also supports an existing abstract machine based on the token-passing GoI, which will be discussed later, to illustrate various resource usage of abstract machines.

This chapter is organised as follows. We present the sub-machine semantics in Sec. 2.2, and introduce the DGoIM with the rewrites-first strategy in Sec. 2.3. In Sec. 2.4, we show how the DGoIM implements the three evaluation strategies via translation of terms into graphs, and establish a weak simulation of the sub-machine semantics by the DGoIM. The simulation result is used to prove soundness and completeness of the DGoIM, and to analyse its time cost, in Sec. 2.5. We compare our graph-rewriting approach to improve time efficiency of token-passing GoI, with another approach from the literature, namely the so-called *jumping* approach, in Sec. 2.6.

[1]Link to the on-line visualiser: `https://koko-m.github.io/GoI-Visualiser/`

## 2.2 A term calculus with sub-machine semantics

We use an untyped term calculus that accommodates three evaluation strategies of the lambda-calculus, by dedicated constructors for function application: namely, @ (call-by-need), $\overrightarrow{@}$ (left-to-right call-by-value) and $\overleftarrow{@}$ (right-to-left call-by-value). Mixing strategies in a single calculus is solely for the purpose of not presenting three almost identical calculi. Even though the term calculus allows one to write a term that uses all strategies, we are not interested in interaction between the three strategies. Instead, our aim is to analyse evaluation cost of lambda-terms with respect to each single strategy. In the rest of the chapter, we assume that each term contains function applications of a single strategy only.

As shown in the top of Fig. 2.1, the calculus accommodates explicit substitutions $[x \leftarrow u]$. A term with no explicit substitutions is said to be *pure*.

The sub-machine semantics is used to establish the soundness of the graph-rewriting abstract machine. It imitates an abstract machine, by having the following two features. Firstly, it extends conventional reduction semantics with reduction steps that explicitly search for a redex, following the style of *token-passing* semantics given by Sinot [2005, 2006]. Secondly, it decomposes the meta-level substitution into on-demand linear substitution, using explicit substitutions, as linear substitution calculi do [Accattoli and Kesner, 2010]. The sub-machine semantics also resembles a storeless abstract machine (e.g. [Danvy et al., 2012, Fig. 8]). However the sub-machine semantics is still too "abstract" to be considered an abstract machine, in the sense that it works modulo alpha-equivalence to avoid variable captures.

Fig. 2.1 defines the sub-machine semantics of our calculus. It is given by labelled relations between *enriched* terms that are in the form of $E\langle(\!|t|\!)\rangle$. In an enriched term $E\langle(\!|t|\!)\rangle$, a sub-term $t$ is not plugged directly into an evaluation context, but into a *window* $(\!|\cdot|\!)$ which makes it syntactically obvious where the reduction context is situated. Note that the term $t$ inside the window can be arbitrary. This means

$$
\begin{array}{rrl}
\text{Terms} & t & ::= x \mid \lambda x.t \mid t \mathbin{@} t \mid t \mathbin{\overrightarrow{@}} t \mid t \mathbin{\overleftarrow{@}} t \mid t[x \leftarrow t] \\
\text{Values} & v & ::= \lambda x.t \\
\text{Answer contexts} & A & ::= \langle\cdot\rangle \mid A[x \leftarrow t] \\
\text{Evaluation contexts} & E & ::= \langle\cdot\rangle \mid E[x \leftarrow t] \mid E\langle x\rangle[x \leftarrow E] \\
 & & \quad \mid E \mathbin{@} t \mid E \mathbin{\overrightarrow{@}} t \mid A\langle v\rangle \mathbin{\overrightarrow{@}} E \mid t \mathbin{\overleftarrow{@}} E \mid E \mathbin{\overleftarrow{@}} A\langle v\rangle
\end{array}
$$

Basic rules $\mapsto_\beta, \mapsto_\sigma, \mapsto_\epsilon$:

$$(\!|t \mathbin{@} u|\!) \mapsto_\epsilon (\!|t|\!) \mathbin{@} u \tag{2.1}$$

$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{@} u \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow u]\rangle \tag{2.2}$$

$$(\!|t \mathbin{\overrightarrow{@}} u|\!) \mapsto_\epsilon (\!|t|\!) \mathbin{\overrightarrow{@}} u \tag{2.3}$$

$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overrightarrow{@}} u \mapsto_\epsilon A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} (\!|u|\!) \tag{2.4}$$

$$A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} A'\langle(\!|v|\!)\rangle \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle \tag{2.5}$$

$$(\!|t \mathbin{\overleftarrow{@}} u|\!) \mapsto_\epsilon t \mathbin{\overleftarrow{@}} (\!|u|\!) \tag{2.6}$$

$$t \mathbin{\overleftarrow{@}} A\langle(\!|v|\!)\rangle \mapsto_\epsilon (\!|t|\!) \mathbin{\overleftarrow{@}} A\langle v\rangle \tag{2.7}$$

$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overleftarrow{@}} A'\langle v\rangle \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle \tag{2.8}$$

$$E\langle(\!|x|\!)\rangle[x \leftarrow A\langle u\rangle] \mapsto_\epsilon E\langle x\rangle[x \leftarrow A\langle(\!|u|\!)\rangle] \quad (u \text{ is not in the form of } A'\langle t'\rangle) \tag{2.9}$$

$$E\langle x\rangle[x \leftarrow A\langle(\!|v|\!)\rangle] \mapsto_\sigma A\langle E\langle(\!|v|\!)\rangle[x \leftarrow v]\rangle \tag{2.10}$$

Reductions $\multimap_\beta, \multimap_\sigma, \multimap_\epsilon$:

$$\frac{\tilde{t} \mapsto_\chi \tilde{u}}{E\langle\tilde{t}\rangle \multimap_\chi E\langle\tilde{u}\rangle} \ (\chi \in \{\beta, \sigma, \epsilon\})$$

Figure 2.1: "Sub-machine" operational semantics

that there may be several enriched terms that have the same underlying ordinary term: namely, $E\langle(\!|t|\!)\rangle$ and $E'\langle(\!|t'|\!)\rangle$ such that $E\langle(\!|t|\!)\rangle \neq E'\langle(\!|t'|\!)\rangle$ as enriched terms but $E\langle t\rangle = E'\langle t'\rangle$ as ordinary terms forgetting the window. This is crucial to explicitly represent the redex search process in the sub-machine semantics, as a move of the window on the same ordinary term.

Basic rules $\mapsto$ are labelled with $\beta$, $\sigma$ or $\epsilon$. The basic rules (2.2), (2.5) and (2.8), labelled with $\beta$, apply beta-reduction and delay substitution for a bound variable. Substitution is done one by one, and on demand, by the basic rule (2.10) with label $\sigma$. Each application of the basic rule (2.10) replaces exactly one occurrence

of a bound variable with a value, and keeps a copy of the value for later use. Note that the basic rule (2.10) does not duplicate the answer context ($A$ in Fig. 2.1) that used to accompany the substituted value. The answer context is instead left shared between two copies of the value, which makes sure that only one value is duplicated in each application of the basic rule (2.10). All other basic rules, with label $\epsilon$, search for a redex by moving the window without changing the underlying term.

Reduction is defined by congruence of basic rules with respect to evaluation contexts, and labelled accordingly. Any basic rules and reductions are indeed between enriched terms, because the window $(\!|\cdot|\!)$ is never duplicated or discarded. They are also deterministic.

An *evaluation* of a pure term $t$ (i.e. a term with no explicit substitution) is a sequence of reductions starting from $\langle (\!|t|\!) \rangle$, which is simply $(\!|t|\!)$. Fig. 2.2 shows evaluations of a pure term $(\lambda x.x)\,((\lambda y.y)\,(\lambda z.z))$ in the three evaluation strategies. Reductions labelled with $\beta$ and $\sigma$, which change an underlying term, are highlighted in black. All three evaluations involve two beta-reductions, which apply $\lambda x.x$ and $\lambda y.y$ to an argument. Application of $\lambda x.x$ comes first in the call-by-need evaluation, and delayed application of $\lambda y.y$ happens inside an explicit substitution. On the other hand, in two call-by-value evaluations, application of $\lambda y.y$ comes first, and no reduction happens inside an explicit substitution. The two call-by-value evaluations differ only in the way the window is moved around function application.

The following lemma enables us to follow the use of sub-terms of the initial term $t$ during the evaluation.

**Lemma 2.2.1.** *For any evaluation $(\!|t|\!) \multimap^* E'\langle (\!|t'|\!) \rangle$ starting from a pure closed term $t$, the term $t'$ is a sub-term of $t$. Moreover, the evaluation context $E'$ is given by the*

Call-by-need evaluation:

$$
\begin{aligned}
(\!|(\lambda x.x) \mathbin{@} ((\lambda y.y) \mathbin{@} (\lambda z.z))|\!) &\multimap_\epsilon (\!|\lambda x.x|\!) \mathbin{@} ((\lambda y.y) \mathbin{@} (\lambda z.z)) \\
&\multimap_\beta (\!|x|\!)[x \leftarrow (\lambda y.y) \mathbin{@} (\lambda z.z)] \\
&\multimap_\epsilon x[x \leftarrow (\!|(\lambda y.y) \mathbin{@} (\lambda z.z)|\!)] \\
&\multimap_\epsilon x[x \leftarrow (\!|\lambda y.y|\!) \mathbin{@} (\lambda z.z)] \\
&\multimap_\beta x[x \leftarrow (\!|y|\!)[y \leftarrow \lambda z.z]] \\
&\multimap_\epsilon x[x \leftarrow y[y \leftarrow (\!|\lambda z.z|\!)]] \\
&\multimap_\sigma x[x \leftarrow (\!|\lambda z.z|\!)[y \leftarrow \lambda z.z]] \\
&\multimap_\sigma (\!|\lambda z.z|\!)[x \leftarrow \lambda z.z][y \leftarrow \lambda z.z]
\end{aligned}
$$

Call-by-value evaluations:

$$
\begin{aligned}
&(\!|(\lambda x.x) \overrightarrow{@} ((\lambda y.y) \overrightarrow{@} (\lambda z.z))|\!) \\
&\multimap_\epsilon (\!|\lambda x.x|\!) \overrightarrow{@} ((\lambda y.y) \overrightarrow{@} (\lambda z.z)) \\
&\multimap_\epsilon \lambda x.x \overrightarrow{@} (\!|(\lambda y.y) \overrightarrow{@} (\lambda z.z)|\!) \\
&\multimap_\epsilon \lambda x.x \overrightarrow{@} ((\!|\lambda y.y|\!) \overrightarrow{@} (\lambda z.z)) \\
&\multimap_\epsilon \lambda x.x \overrightarrow{@} ((\lambda y.y) \overrightarrow{@} (\!|\lambda z.z|\!)) \\
&\multimap_\beta \lambda x.x \overrightarrow{@} (\!|y|\!)[y \leftarrow \lambda z.z]) \\
&\multimap_\epsilon \lambda x.x \overrightarrow{@} (y[y \leftarrow (\!|\lambda z.z|\!)]) \\
&\multimap_\sigma \lambda x.x \overrightarrow{@} (\!|\lambda z.z|\!)[y \leftarrow \lambda z.z]) \\
&\multimap_\beta (\!|x|\!)[x \leftarrow (\lambda z.z)[y \leftarrow \lambda z.z]] \\
&\multimap_\epsilon x[x \leftarrow (\!|\lambda z.z|\!)[y \leftarrow \lambda z.z]] \\
&\multimap_\sigma (\!|\lambda z.z|\!)[x \leftarrow \lambda z.z][y \leftarrow \lambda z.z]
\end{aligned}
\qquad
\begin{aligned}
&(\!|(\lambda x.x) \overleftarrow{@} ((\lambda y.y) \overleftarrow{@} (\lambda z.z))|\!) \\
&\multimap_\epsilon \lambda x.x \overleftarrow{@} (\!|(\lambda y.y) \overleftarrow{@} (\lambda z.z)|\!) \\
&\multimap_\epsilon \lambda x.x \overleftarrow{@} ((\lambda y.y) \overleftarrow{@} (\!|\lambda z.z|\!)) \\
&\multimap_\epsilon \lambda x.x \overleftarrow{@} ((\!|\lambda y.y|\!) \overleftarrow{@} (\lambda z.z)) \\
&\multimap_\beta \lambda x.x \overleftarrow{@} (\!|y|\!)[y \leftarrow \lambda z.z]) \\
&\multimap_\epsilon \lambda x.x \overleftarrow{@} (y[y \leftarrow (\!|\lambda z.z|\!)]) \\
&\multimap_\sigma \lambda x.x \overleftarrow{@} (\!|\lambda z.z|\!)[y \leftarrow \lambda z.z]) \\
&\multimap_\epsilon (\!|\lambda x.x|\!) \overleftarrow{@} ((\lambda z.z)[y \leftarrow \lambda z.z]) \\
&\multimap_\beta (\!|x|\!)[x \leftarrow (\lambda z.z)[y \leftarrow \lambda z.z]] \\
&\multimap_\epsilon x[x \leftarrow (\!|\lambda z.z|\!)[y \leftarrow \lambda z.z]] \\
&\multimap_\sigma (\!|\lambda z.z|\!)[x \leftarrow \lambda z.z][y \leftarrow \lambda z.z]
\end{aligned}
$$

Figure 2.2: Evaluations of $(\lambda x.x)\,((\lambda y.y)\,(\lambda z.z))$

*following restricted grammar:*

$$
\begin{aligned}
\overline{A} &::= \langle\cdot\rangle \mid \overline{A}[x \leftarrow \overline{A}\langle u\rangle], \\
\overline{E} &::= \langle\cdot\rangle \mid \overline{E}[x \leftarrow \overline{A}\langle u\rangle] \mid \overline{E}\langle x\rangle[x \leftarrow \overline{E}] \\
&\quad \mid \overline{E} \mathbin{@} u \mid \overline{E} \overrightarrow{@} u \mid \overline{A}\langle v\rangle \overrightarrow{@} \overline{E} \mid u \overleftarrow{@} \overline{E} \mid \overline{E} \overleftarrow{@} \overline{A}\langle v\rangle
\end{aligned}
$$

*where $u$ and $v$ are sub-terms of $t$, and $v$ is additionally a value.*

*Proof outline.* The proof is by induction on the length $k$ of the evaluation $(\!|t|\!) \multimap^k$

$E'\langle(\!|t'|\!)\rangle$. In the base case, where $k = 0$, we have $\overline{E} = \langle\cdot\rangle$ and $t' = t$. The inductive case, where $k > 0$, is proved by inspecting a basic rule used in the last reduction of the evaluation. In the case of the basic rule (2.9), the last reduction is in the form of $E_0\langle E\langle(\!|x|\!)\rangle[x \leftarrow A\langle u\rangle]\rangle \multimap_\epsilon E_0\langle E\langle x\rangle[x \leftarrow A\langle(\!|u|\!)\rangle]\rangle$ where $u$ is not in the form of $A''\langle t''\rangle$. By induction hypothesis, $E_0\langle E\langle\rangle[x \leftarrow A\langle u\rangle]\rangle$ follows the restricted grammar, and in particular, $A\langle u\rangle$ can be decomposed into a restricted answer context and a sub-term of $t$. Because a sub-term of $t$ is also pure, it follows that $A$ itself is a restricted answer context and $u$ is a sub-term of $t$. □

## 2.3 The token-guided graph-rewriting machine

In the initial presentation of this work [Muroya and Ghica, 2017], we used proof nets of the multiplicative and exponential fragment of linear logic [Girard, 1987] to implement the call-by-need evaluation strategy. Aiming additionally at two call-by-value evaluation strategies, we here use graphs that are closer to syntax trees but are still augmented with the !-box structure taken from proof nets. Moving towards syntax trees allows us to implement two call-by-value evaluations in a uniform way. The !-box structure specifies duplicable sub-graphs, and help time-cost analysis of implementations.

### 2.3.1 Graphs with interface

We use directed graphs, whose nodes are classified into *proper* nodes and *link* nodes. Link nodes are required to meet the following conditions.

- For each edge, at least one of its two endpoints is a link node.
- Each link node is a source of at most one edge, and a target of at most one edge.

In particular, a link node is called *input* if it is not a target of any edge, and *output*

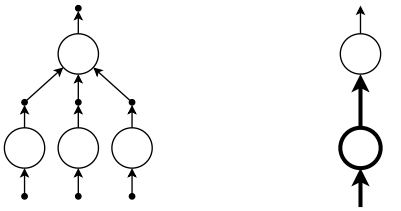


Figure 2.3: Full (left) and simplified (right) representation of a graph $G(3, 1)$

if it is not a source of any edge.[2] An *interface* of a graph is given by the set of all inputs and the set of all outputs. When a graph $G$ has $n$ input link nodes and $m$ output link nodes, we sometimes write $G(n, m)$ to emphasise its interface. If a graph has exactly one input, we refer to the input link node as *root*.

An example graph $G(3, 1)$ is shown on the left in Fig. 2.3. It has four proper nodes depicted by circles, and seven link nodes depicted by bullets. Its three inputs are placed at the bottom and one output is at the top. Shown on the right in Fig. 2.3 is a simplified version of the representation. We use the following simplification scheme: not drawing link nodes explicitly (unless necessary), and using a bold-stroke arrow (resp. circle) to represent a bunch of parallel edges (resp. proper nodes).

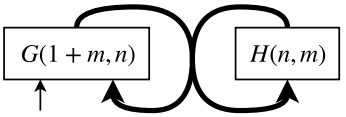
The idea of using link nodes, as distinguished from proper nodes, comes from a graphical formalisation of string diagrams [Kissinger, 2012].[3] String diagrams consist of *boxes* that are connected to each other by *wires*, and may have dangling or looping wires. In the formalisation, boxes are modelled by *box-vertices* (corresponding to proper nodes in our case), and wires are modelled by consecutive edges connected via *wire-vertices* (corresponding to link nodes in our case). It is link nodes that allow dangling or looping wires to be properly modelled. The segmentation of wires into edges can introduce an arbitrary number of consecutive link nodes, however these consecutive link nodes are identified by the notion of *wire homeomorphism*. We will later discuss these consecutive link nodes, from the perspective of the graph-

[2] In graph-theoretical terminology, *source* means what we call *input*, and *sink* means what we call *output*. Our terminology is to avoid the abuse of the term "source" that refers to one endpoint of a directed edge.

[3] Our link nodes should not be confused with the same terminology "link" of proof nets, which refers to a counterpart of our proper nodes.

rewriting machine. From now on we simply call a proper node "node", and a link node "link".

Finally, an operation $\circ_{n,m}$ on graphs, parametrised by natural numbers $n$ and $m$, is defined as follows:

$G \circ_{n,m} H :=$ 

.

In the sequel, we omit the parameters $n, m$ and simply write $\circ$.

### 2.3.2 Node labels and !-boxes

We use the following set $\mathcal{L}$ to label nodes:

$$\mathcal{L} = \{\lambda, @, \overrightarrow{@}, \overleftarrow{@}, !, ?, D\} \cup \{C_n \mid n\text{: a natural number}\}.$$

A node labelled with $X \in \mathcal{L}$ is called an *X-node*. The first four labels correspond to the constructors of the calculus presented in Sec. 2.2, namely $\lambda$ (abstraction), @ (call-by-need application), $\overrightarrow{@}$ (left-to-right call-by-value application) and $\overleftarrow{@}$ (right-to-left call-by-value application). These three application nodes are a part of the novelty of this work. The token, travelling in a graph, reacts to these nodes in different ways, and hence implements different evaluation orders. We believe that it is a more extensible way to accommodate different evaluation orders as different nodes, than to let the token react to the same node in different ways depending on situation. The other labels, namely !, ?, $D$ and $C_n$ for any natural number $n$, are used in the management of copying sub-graphs. These are inspired by proof nets of the multiplicative and exponential fragment of linear logic [Girard, 1987], and $C_n$-nodes generalise the standard binary contraction and subsume weakening.

We use the *generators* in Fig. 2.4 to build labelled graphs. Most generators are given by a graph that consists of one labelled node and a fixed number of interface

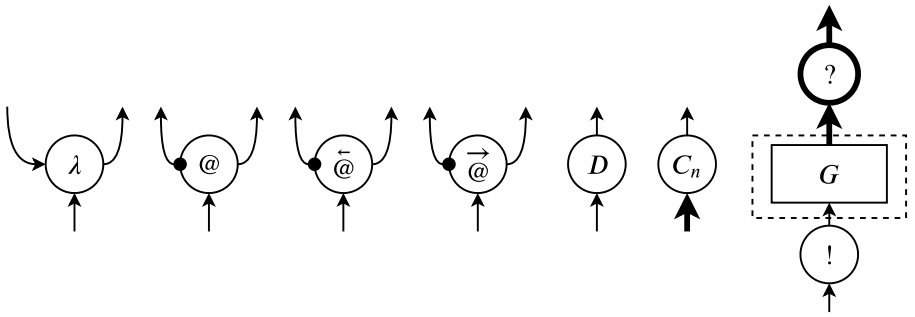



Figure 2.4: Generators of graphs

links, which are adjacent to the node. The label of the node determines the interface links and their connection with the node, as indicated in the figure. For example, a label $\lambda$ indicates three edges connecting a $\lambda$-node with one input link and two output links. Going clockwise from the bottom, they are: one edge from the input link, one from an output link, and one to an output link. Application generators (@, $\overrightarrow{@}$ or $\overleftarrow{@}$) have one edge from an input link and two edges to output links. We distinguish the two output links, calling one "*function output*" and the other "*argument output*" (cf. [Accattoli and Guerrini, 2009]). A bullet $\bullet$ in the figure specifies an edge to a function output. A label $C_n$ indicates $n$ incoming edges from $n$ input links and one outgoing edge to an output link.

The last generator in Fig. 2.4 turns a graph $G(1, m)$ into a sub-graph (!-*box*), by connecting it to one !-node (*principal door*) and $m$ ?-nodes (*auxiliary doors*). This !-box structure is indicated by a dashed box in the figure. The !-box structure, taken from proof nets, assists the management of duplication of sub-graphs by specifying those that can be copied.[4]

### 2.3.3 Graph states and transitions

We define a graph-rewriting abstract machine as a labelled transition system between *graph states*.

[4]Our formalisation of graphs is related to the view of proof nets as string diagrams, and hence of !-boxes as functorial boxes [Melliès, 2006].

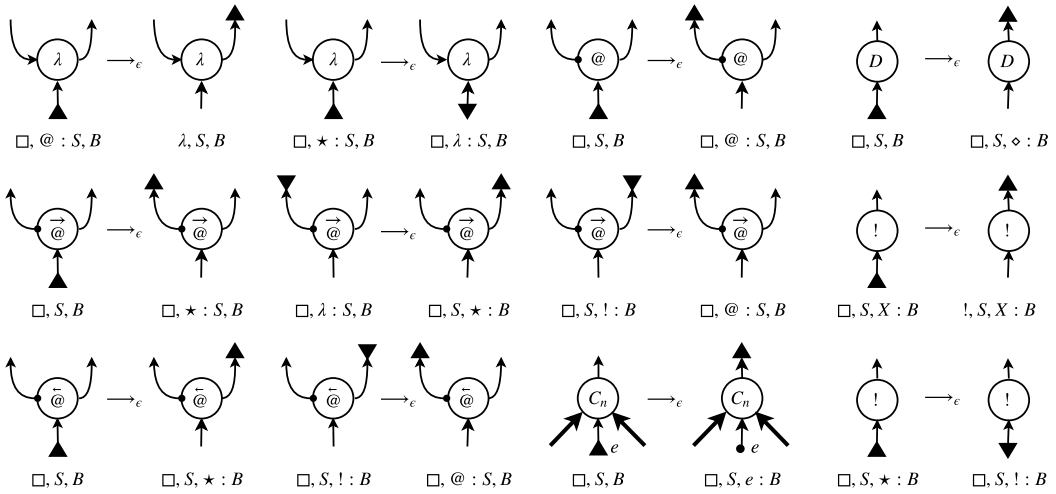
**Definition 2.3.1** (Graph states)**.** A *graph state* $((G(1,0),e),\delta)$ is formed of a graph $G(1,0)$ with its distinguished link $e$, and *token data* $\delta = (d, f, S, B)$ that consists of:

- a *direction* defined by $d ::= \uparrow \mid \downarrow$,
- a *rewrite flag* defined by $f ::= \square \mid \lambda \mid !$,
- a *computation stack* defined by $S ::= \square \mid \star : S \mid \lambda : S \mid @ : S$, and
- a *box stack* defined by $B ::= \square \mid \star : B \mid ! : B \mid \diamond : B \mid e' : B$, where $e'$ is any link of the graph $G$.

The distinguished link $e$ of a graph state $((G,e),(d,f,S,B))$ is called the *position* of the token. Recall that any link of a graph, including the position, has at most one incoming edge and at most one outgoing edge. The position will change along the outgoing edge when the direction of the token is upwards ($d = \uparrow$), and move against the incoming edge if the direction is downwards ($d = \downarrow$).[5] These token moves can only happen when the rewrite flag is not raised, namely when $f = \square$. Otherwise the graph $G$ is rewritten, as instructed by the flag; the rewrite targets a $\lambda$-node when $f = \lambda$, and targets a !-box when $f = !$.

The token uses stacks to determine, and record, its reaction to potential targets of rewrites: namely, it uses the computation stack $S$ for $\lambda$-nodes, and the box stack $B$ for !-boxes. The element '$\star$' at the top of either stack instructs the token not to perform a rewrite even if the token finds a $\lambda$-node or a !-box. Instead, a new element is placed at the top of the stack: namely, '$\lambda$' indicating the $\lambda$-node or '!' indicating the !-box. Any other elements at the top of the stacks enable the token to actually trigger a rewrite. They also help the token determine which rewrite to trigger, by indicating a node to be involved in the rewrite. These elements are namely: '@' of the computation stack indicating an application node (i.e. nodes labelled with @, $\overrightarrow{@}$

[5] The way the token direction works is tailored to our drawing convention of graphs, which is to draw directed edges mostly upwards.




where $X \neq \star$.

Figure 2.5: Pass transitions

or $\overleftarrow{@}$), '$\diamond$' of the box stack indicating a $D$-node, and a link of the graph $G$ indicating a $C$-node whose inputs include the link.

**Definition 2.3.2** (Initial/final states)**.**

1. A state $((G, e_0), (\uparrow, \square, \square, \star : \square))$, where $e_0$ is the root of the graph $G(1, 0)$, is said to be *initial*.

2. A state $((G, e_0), (\downarrow, \square, \square, ! : \square))$, where $e_0$ is the root of the graph $G(1, 0)$, is said to be *final*.

By the above definition, any graph $G(1, 0)$ uniquely induces an initial state, denoted by $Init(G)$, and a final state, denoted by $Final(G)$. An *execution* on a graph $G$ is a sequence of transitions starting from the initial state $Init(G)$.

Each transition $((G, e), \delta) \rightarrow_\chi ((G', e'), \delta')$ between graph states is labelled by either $\beta$, $\sigma$ or $\epsilon$. Transitions are deterministic, and classified into *pass* transitions that search for redexes and trigger rewriting, and *rewrite* transitions that actually rewrite a graph as soon as a redex is found.

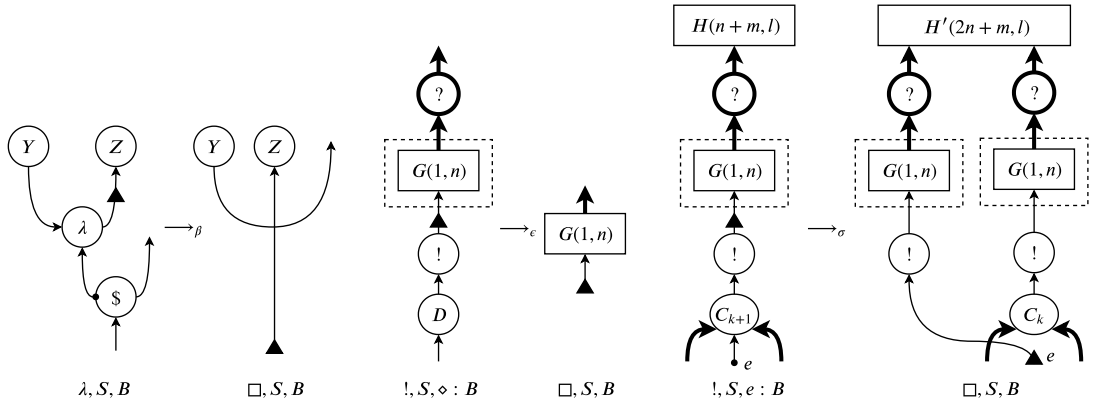
A pass transition $((G \circ H, e), (d, \square, S, B)) \rightarrow_\epsilon ((G \circ H, e'), (d', f', S', B'))$, always labelled with $\epsilon$, applies to a state whose rewrite flag is $\square$. The graph $H$ contains only

one node, and the positions $e$ and $e'$ are an input or an output of the node. Fig. 2.5 defines pass transitions, by showing the single-node graph $H$, token positions and data, omitting the irrelevant graph $G$. The position of the token is drawn as a black triangle, pointing towards the direction of the token.

Each pass transition simply moves the token over a node, and modifies the token data, while keeping an underlying graph unchanged. When the token passes a $\lambda$-node or a !-node, a rewrite flag is changed to $\lambda$ or !, which triggers a rewrite transition. When the token passes a $C_n$-node, where $n$ is positive, the old position $e$ is pushed to a box stack. This link $e$ is drawn as a bullet in Fig. 2.5.

The way the token reacts to application nodes (@, $\overrightarrow{@}$ and $\overleftarrow{@}$) corresponds to the way the window $(\!|\cdot|\!)$ moves in evaluating these function applications in the sub-machine semantics (Fig. 2.1). When the token moves on to the function output of an application node, the top element of a computational stack is either @ or $\star$. The element $\star$ makes the token return from a $\lambda$-node, which corresponds to reducing the function part of application to a value (i.e. abstraction). The element @ lets the token proceed at a $\lambda$-node, raises the rewrite flag $\lambda$, and hence triggers a rewrite transition that corresponds to beta-reduction. The call-by-value application nodes ($\overrightarrow{@}$ and $\overleftarrow{@}$) send the token to their argument output, pushing the element $\star$ to a box stack. This makes the token bounce at a !-node and return to the application node, which corresponds to evaluating the argument part of function application to a value. Finally, pass transitions through $D$-nodes, $C_n$-nodes and !-nodes prepare copying of values, and eventually raise the rewrite flag ! that triggers on-demand duplication.

A rewrite transition $((G \circ H, e), (d, f, S, B)) \rightarrow_\chi ((G \circ H', e'), (d', f', S, B'))$, labelled with $\chi \in \{\beta, \sigma, \epsilon\}$, applies to a state whose rewrite flag is either $\lambda$ or !. It replaces the sub-graph $H$ (*redex*) with the graph $H'$ of the same interface. The position $e$ that belongs to $H$ is changed to the position $e'$ that belongs to $H'$. The transition may pop an element from a box stack. Fig. 2.6 defines rewrite transi-




where $Y \in \mathcal{L}$, $Z \in \mathcal{L}$, $\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$, and $G(1, n)$ is any graph.

Figure 2.6: Rewrite transitions

tions, by showing the sub-graphs $H$ and $H'$, as well as token positions and data, omitting the graph $G$. Before we go through each rewrite transition, we note that rewrite transitions are not exhaustive in general, as a graph may not match a redex even though a rewrite flag is raised. However we will see that there is no failure of transitions in implementing the term calculus.

The first rewrite transition in Fig. 2.6, with label $\beta$, occurs when a rewrite flag is $\lambda$. It implements beta-reduction by eliminating a pair of an abstraction node ($\lambda$) and an application node ($\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$ in the figure). Outputs of the $\lambda$-node are required to be connected to arbitrary nodes (labelled with $Y$ and $Z$ in the figure), so that edges between links are not introduced. The $Y$-node and the $Z$-node may be the same node.

The other rewrite transitions in Fig. 2.6 are for the rewrite flag !, and they target at duplicable sub-graphs, i.e. !-boxes. They also pop the top element of a box stack, which is used to determine which rewrite to perform.

The second rewrite transition in the figure, labelled with $\epsilon$, finishes off each duplication process by *opening* the !-box $G$. This box-opening operation eliminates all doors of the !-box $G$, and replaces the interface of $G$ with output links of the auxiliary doors and the input link of the $D$-node, which is the new position of the token. Again, no edge between links are introduced.

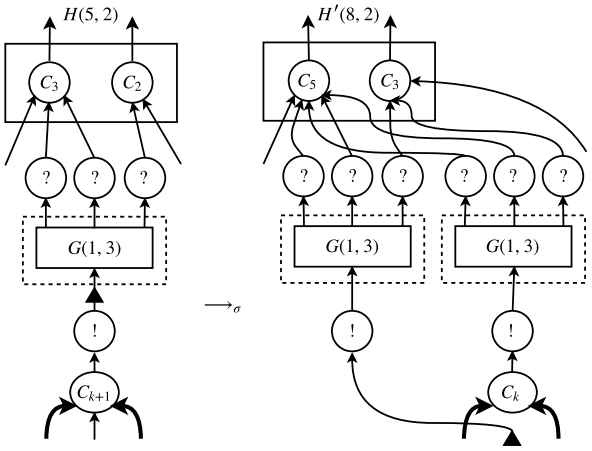



Figure 2.7: Example of rewrite transition $\rightarrow_\sigma$

The last rewrite transition in the figure, with label $\sigma$, actually copies a !-box. It requires the top element $e$ of the old box stack to be one of input links of the $C_{k+1}$-node (where $k$ is a natural number). The link $e$ is popped from the box stack and becomes the new position of the token, and the $C_{k+1}$-node becomes a $C_k$-node by keeping all the inputs except for the link $e$. The sub-graph $H(n+m, l)$ must consist of $l$ parallel $C$-nodes that altogether have $n+m$ inputs. Among these inputs, $n$ must be connected to auxiliary doors of the !-box $G(1, n)$, and $m$ must be connected to nodes that are not in the redex. The sub-graph $H(n+m, l)$ is turned into $H'(2n+m, l)$ by introducing $n$ inputs to these $C$-nodes as follows: if an auxiliary door of the !-box $G$ is connected to a $C$-node in $H$, two copies of the auxiliary door are both connected to the corresponding $C$-node in $H'$. Therefore the two sub-graphs consist of the same number $l$ of $C$-nodes, whose in-degrees are possibly increased. The $m$ inputs, connected to nodes outside a redex, are kept unchanged. Fig. 2.7 shows an example where copying of the graph $G(1, 3)$ turns the graph $H(5, 2)$ into $H'(8, 2)$.

All pass and rewrite transitions are well-defined, and indeed deterministic. Pass transitions are also reversible, in the sense that no two different pass transitions result in the same graph state. No transition is possible at a final state, and no pass transition results in an initial state. Fig. 2.8 shows an example execution of the DGoIM, which starts from an initial state and terminates at a final state. As

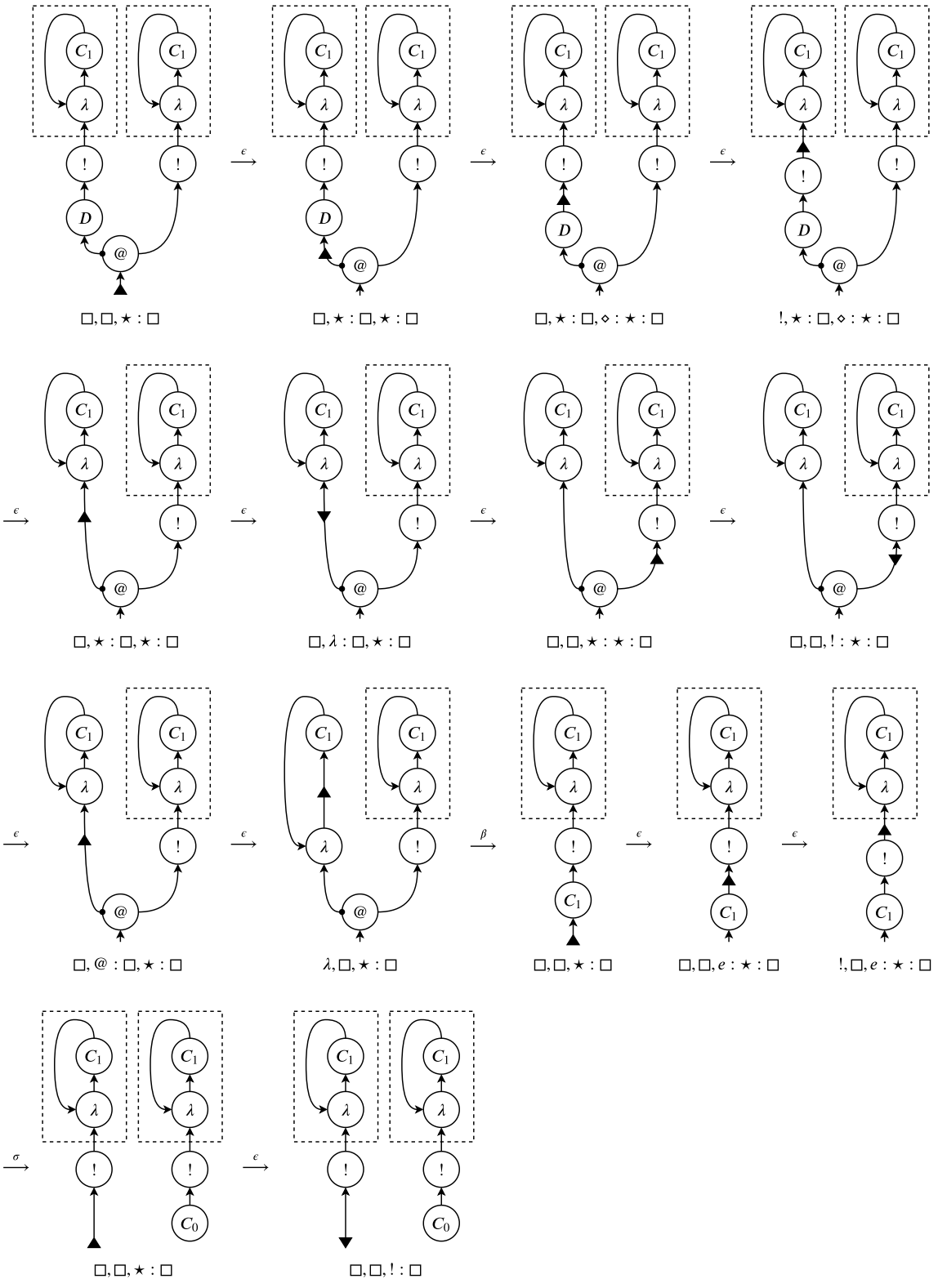



Figure 2.8: Example execution of the DGoIM

will be apparent in Sec. 2.4, this execution corresponds to evaluation of a term $(\lambda x.x) \overrightarrow{@} (\lambda y.y)$.

An execution of only pass transitions has some continuity in the following sense.

**Lemma 2.3.3** (Pass continuity)**.** *For any execution* $Init(G) \to^* ((G, e), \delta)$ *of pass transitions only, there exists a non-empty sequence* $e_1, \ldots, e_n$ *of links of* $G$ *that satisfies the following.*

- $e_1$ *is the root of* $G$*, and* $e_n = e$*.*
- *For each* $i \in \{1, \ldots, n-1\}$*, there exists a node whose inputs include* $e_i$ *and whose outputs include* $e_{i+1}$*.*
- *Each link in the sequence appears as a token position in the execution* $Init(G) \to^* ((G, e), \delta)$*.*

*Proof outline.* The proof is by induction on the length $k$ of the execution $Init(G) \to^* ((G, e), \delta)$. In the base case, where $k = 0$, the link $e$ is the root of $G$, and $e$ itself as a sequence satisfies the conditions. The inductive case, where $k > 0$, is proved by inspecting all possibilities of the last pass transition in the sequence. □

The following *sub-graph* property is essential in time-cost analysis, because it bounds the size of duplicable sub-graphs (i.e. !-boxes) in an execution.

**Lemma 2.3.4** (Sub-graph property)**.** *For any execution* $Init(G) \to^* ((H, e), \delta)$*, each* !*-box of the graph* $H$ *appears as a sub-graph of the initial graph* $G$*.*

*Proof.* Rewrite transitions can only copy or discard a !-box, and cannot introduce, expand or reduce a single !-box. Therefore, any !-box of $H$ has to be already a !-box of the initial graph $G$. □

When a graph has an edge between links, the token is just passed along. With this pass transition over a link at hand, the equivalence relation between graphs that identifies consecutive links with a single link—so-called *wire homeomorphism* [Kissinger,

2012]—lifts to a weak bisimulation between graph states. Therefore, behaviourally, we can safely ignore consecutive links. From the perspective of time-cost analysis, we benefit from the fact that rewrite transitions can be performed without introducing any edge between links; in other words, any edges between links introduced by a rewrite transition can be immediately eliminated by identifying endpoints. This means that, by assuming that an execution starts with a graph with no consecutive links, we can analyse time cost of the execution without caring the extra pass transition over a link.

## 2.4 Implementation of evaluation strategies

The implementation of the term calculus, by means of the dynamic GoI, starts with translating (enriched) terms into graphs. The definition of the translation uses multisets of variables, to track how many times each variable occurs in a term. A multiset of variables is given by a function $M : \mathbb{V} \to \mathbb{N}$ from the set of variables to the set of natural numbers, such that only a finite number of variables are mapped to positive numbers. We assume that terms are alpha-converted in a form in which all binders introduce distinct variables.

**Notation 1** (Multiset)**.** *We write $x \in^k M$ if $M(x) = k$, that is, the multiplicity of $x$ in a multiset $M$ is $k$. The empty multiset is denoted by $\emptyset$, which means $\emptyset(x) = 0$ for any $x$. The sum of two multisets $M_1$ and $M_2$, denoted by $M_1 + M_2$, is defined by $(M_1 + M_2)(x) = M_1(x) + M_2(x)$. We can remove* all *occurrences of $x$ from a multiset $M$ by changing the multiplicity of $x$ to zero. This yields the multiset $M\backslash x$, e.g. $[x, x, y]\backslash x = [y]$. We abuse the notation and refer to a multiset $[x, \ldots, x]$ of a finite number of $x$'s, simply as $x$.*

**Definition 2.4.1** (Free variables)**.** The map FV of terms to multisets of variables

is inductively defined as below, where $\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$:

$$\begin{aligned}
\mathrm{FV}(x) &:= [x], \\
\mathrm{FV}(\lambda x.t) &:= \mathrm{FV}(t) \backslash x, \\
\mathrm{FV}(t \mathbin{\$} u) &:= \mathrm{FV}(t) + \mathrm{FV}(u), \\
\mathrm{FV}(t[x \leftarrow u]) &:= (\mathrm{FV}(t) \backslash x) + \mathrm{FV}(u).
\end{aligned}$$

For a multiset $M$ of variables, the map $\mathrm{FV}_M$ of evaluation contexts to multisets of variables is defined by:

$$\begin{aligned}
\mathrm{FV}_M(\langle\cdot\rangle) &:= M, \\
\mathrm{FV}_M(E @ t) &:= \mathrm{FV}_M(E) + \mathrm{FV}(t), \\
\mathrm{FV}_M(E \overrightarrow{@} t) &:= \mathrm{FV}_M(E) + \mathrm{FV}(t), \\
\mathrm{FV}_M(A\langle v\rangle \overrightarrow{@} E) &:= \mathrm{FV}(A\langle v\rangle) + \mathrm{FV}_M(E), \\
\mathrm{FV}_M(t \overleftarrow{@} E) &:= \mathrm{FV}(t) + \mathrm{FV}_M(E), \\
\mathrm{FV}_M(E \overleftarrow{@} A\langle v\rangle) &:= \mathrm{FV}_M(E) + \mathrm{FV}(A\langle v\rangle), \\
\mathrm{FV}_M(E[x \leftarrow t]) &:= (\mathrm{FV}_M(E) \backslash x) + \mathrm{FV}(t), \\
\mathrm{FV}_M(E'\langle x\rangle[x \leftarrow E]) &:= (\mathrm{FV}(E'\langle x\rangle) \backslash x) + \mathrm{FV}_M(E).
\end{aligned}$$

A term $t$ is said be *closed* if $\mathrm{FV}(t) = \emptyset$. Consequences of the above definition are the following equations.

$$\begin{aligned}
\mathrm{FV}(E\langle t\rangle) &= \mathrm{FV}_{\mathrm{FV}(t)}(E), \\
\mathrm{FV}_M(E\langle E'\rangle) &= \mathrm{FV}_{\mathrm{FV}_M(E')}(E), \\
\mathrm{FV}_{M+M'}(E) &= \mathrm{FV}_M(E) + M' \quad (\text{if } M' \text{ is not captured in } E), \\
\mathrm{FV}_x(E) \backslash x &= \mathrm{FV}_\emptyset(E) \backslash x.
\end{aligned}$$

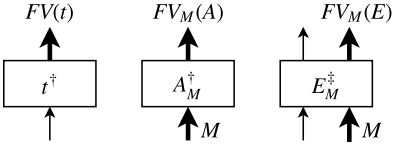



Figure 2.9: General form of translations

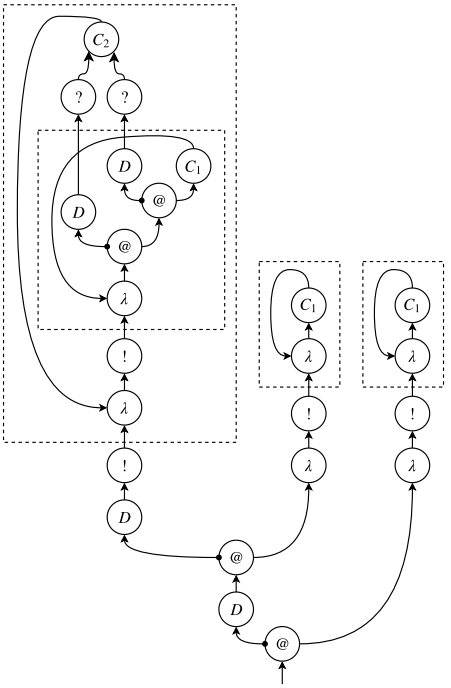



Figure 2.10: Translation of a term $((\lambda f.\lambda x.f \mathbin{@} (f \mathbin{@} x)) \mathbin{@} (\lambda y.y)) \mathbin{@} (\lambda z.z)$

We give translations of terms, answer contexts, and evaluation contexts separately. Fig. 2.11 and Fig. 2.12 define two mutually recursive translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$, the first one for terms and answer contexts, and the second one for evaluation contexts. In the figures, $\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$, and $m$ is the multiplicity of $x$. Fig. 2.9 shows the general form of the translations, and Fig. 2.10 shows translation of a term $((\lambda f.\lambda x.f \mathbin{@} (f \mathbin{@} x)) \mathbin{@} (\lambda y.y)) \mathbin{@} (\lambda z.z)$.

The DGoIM can evaluate a closed term $t$ by starting an execution on the translation $t^\dagger$. The execution shown in Fig. 2.8 is indeed on the translation $((\lambda x.x) \overrightarrow{@} (\lambda y.y))^\dagger$, and executions on any translated closed pure terms can be seen in our on-line visualiser[6]. The translations of answer contexts and evaluation contexts will be used to define a weak simulation between the sub-machine semantics and the DGoIM, both seen as labelled transition systems. The weak simulation plays a key role in proving soundness, completeness and efficiency of the DGoIM.

The annotation of bold-stroke edges means each edge of a bunch is labelled with

[6] `https://koko-m.github.io/GoI-Visualiser/`

an element of the annotating multiset, in a one-to-one manner. In particular if a bold-stroke edge is annotated by a variable $x$, all edges in the bunch are annotated by the variable $x$. Translation $E^{\ddagger}_{M}$ of an evaluation context has one input and one output that are not annotated, which we refer to as the *main input* and the *main output*. These annotations are only used to define the translations, and are subsequently ignored during execution.

The translations are based on the so-called *call-by-value translation* of linear logic to intuitionistic logic (e.g. [Maraist et al., 1999]). It is only abstraction that is translated as a !-box, which captures the fact that only values (i.e. abstractions) can be duplicated (see the basic rule (2.10) in Fig. 2.1). Indeed, if a term $u$ stored in an explicit substitution $[x \leftarrow u]$ is not a value, its translation is not a !-box, and it cannot be duplicated as a whole. Note that only one $C$-node is introduced for each bound variable. This is vital to achieve constant cost in looking up a variable, namely in realising the basic rule (2.9) in Fig. 2.1.

The two mutually recursive translations $(\cdot)^{\dagger}$ and $(\cdot)^{\ddagger}$ are related by the decompositions in Fig. 2.13, which can be checked by straightforward induction. In the third decomposition, $M'$ is not captured in $E$. Note that, in general, the translation $E\langle t\rangle^{\dagger}$ of a term in an evaluation context cannot be decomposed into translations $E^{\ddagger}_{\mathrm{FV}(t)}$ and $t^{\dagger}$. This is because a translation $(A\langle \lambda x.t\rangle \overrightarrow{@}\, E)^{\ddagger}_{M}$ lacks a !-box structure, compared to a translation $(A\langle \lambda x.t\rangle \overrightarrow{@}\, u)^{\dagger}$.

Translation of an evaluation context can be traversed by pass transitions without raising the rewrite flag $\lambda$ or !, as the following lemma states.

**Lemma 2.4.2.** *Let $E$ be an evaluation context and $M$ be a multiset. For any graph $G(1,0)$ that has $E^{\ddagger}_{M}$ as a sub-graph and has no edge between links, let $e_i$ and $e_o$ be the main input and the main output of the sub-graph $E^{\ddagger}_{M}$, respectively. For any pair $(S,B)$ of a computation stack and a box stack, there exists a pair $(S',B')$ of a computation stack and a box stack, such that $((G,e_i),(\uparrow,\Box,S,B)) \rightarrow^{*} ((G,e_o),(\uparrow,\Box,S',B'))$ is a sequence of pass transitions.*

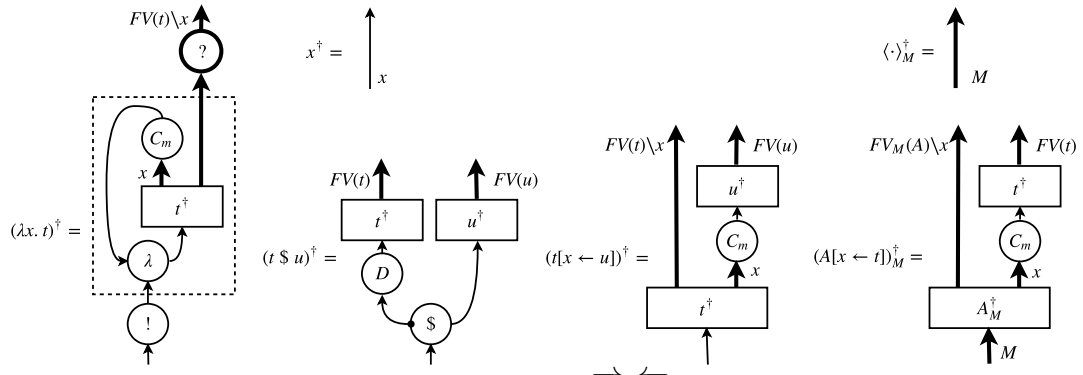



Figure 2.11: Inductive translation of terms and answer contexts

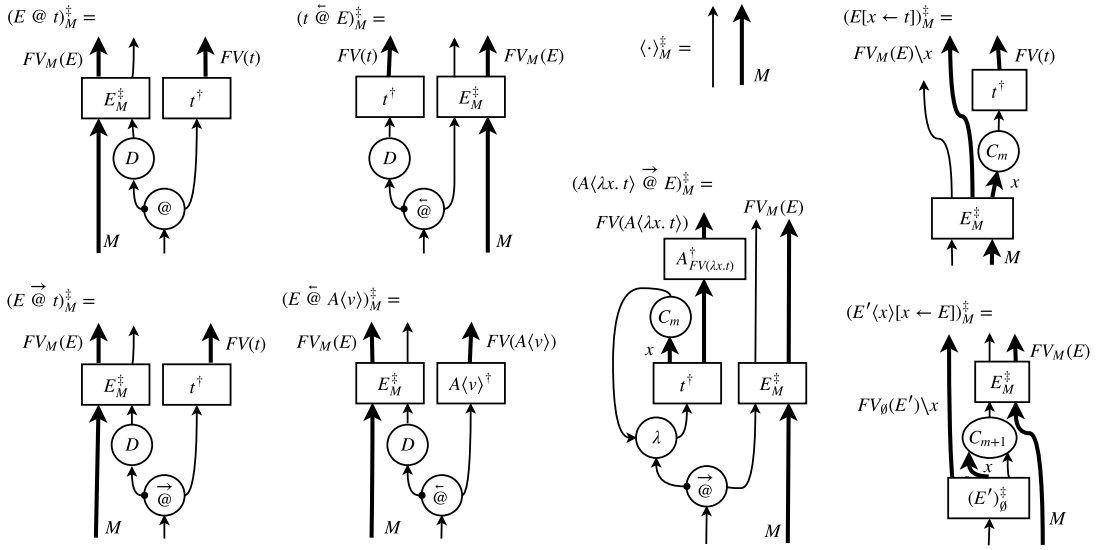



Figure 2.12: Inductive translation of evaluation contexts

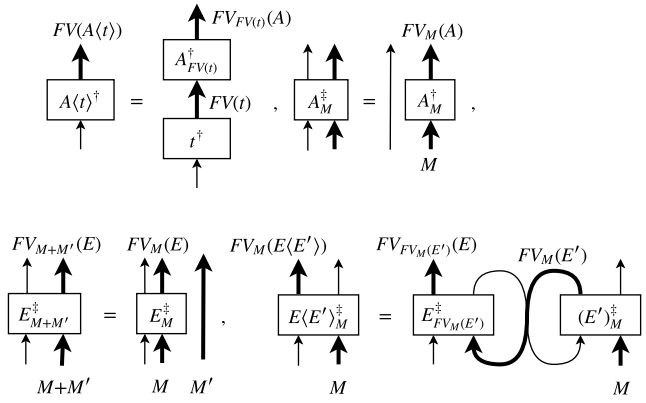



Figure 2.13: Decompositions of translations

*Proof.* By induction on $E$. We use $\xrightarrow{p}{}^*$ to denote a sequence of pass transitions in this proof. In the base case, where $E = \langle\cdot\rangle$, the main input $e_i$ and the main output $e_o$ coincides. An empty sequence suffices.

The first class of inductive cases are when the top-level constructor of $E$ is function application, e.g. $E \equiv E' \mathbin{@} t$. Let $e'_i$ and $e'_o$ be the main input and the main output of the sub-graph $(E')^{\ddagger}_{M}$, respectively. In each of the cases, there exist stacks $S''$ and $B''$ such that $((G, e_i), (\uparrow, \square, S, B)) \xrightarrow{p}{}^* ((G, e'_i), (\uparrow, \square, S'', B''))$. By the induction hypothesis, there exist stacks $S'$ and $B'$ such that $((G, e'_i), (\uparrow, \square, S'', B'')) \xrightarrow{p}{}^* ((G, e'_o), (\uparrow, \square, S', B'))$. Combining these two sequences yields a desired sequence, because $e'_o = e_o$.

The inductive case where $E \equiv E'[x \leftarrow t]$ simply boils down to the induction hypothesis.

The last inductive case is when $E \equiv E_1\langle x\rangle[x \leftarrow E_2]$. Let $e'_i$ and $e'_o$ be the main input and the main output of the sub-graph $(E_1)^{\ddagger}_{\emptyset}$, and $e''_i$ and $e''_o$ be the main input and the main output of the sub-graph $(E_2)^{\ddagger}_{M}$, respectively. We have $e_i = e'_i$ and $e_o = e''_o$. The link $e'_o$ is an input of a $C$-node and $e''_i$ is the output of the $C$-node. By the induction hypothesis on $E_1$, there exist stacks $S''$ and $B''$ such that $((G, e'_i), (\uparrow, \square, S, B)) \xrightarrow{p}{}^* ((G, e'_o), (\uparrow, \square, S'', B''))$. This sequence can be followed by a pass transition $((G, e'_o), (\uparrow, \square, S'', B'')) \to ((G, e''_i), (\uparrow, \square, S'', e'_o : B''))$. By the induction hypothesis on $E_2$, there exist stacks $S'$ and $B'$ such that $((G, e''_i), (\uparrow, \square, S'', e'_o : B'')) \xrightarrow{p}{}^* ((G, e''_o), (\uparrow, \square, S', B'))$. Combining all these sequences yields a desired sequence, because $e_i = e'_i$ and $e_o = e''_o$. □

The inductive translations lift to a binary relation between closed enriched terms and graph states.

**Definition 2.4.3** (Binary relation $\preceq$)**.** The binary relation $\preceq$ is defined by $E\langle(\!|t|\!)\rangle \preceq ((E^{\ddagger} \circ t^{\dagger}, e), (\uparrow, \square, S, B))$, where: (i) $E\langle(\!|t|\!)\rangle$ is a closed enriched term, and $(E^{\ddagger} \circ t^{\dagger}, e)$ is given by 

with no edges between links, and (ii) there is an execution

$\mathit{Init}(E^\ddagger \circ t^\dagger) \to^* ((E^\ddagger \circ t^\dagger, e), (\uparrow, \square, S, B))$ of pass transitions only, in which $e$ appears as a token position only in the last state.

A special case is $(\!|t|\!) \preceq \mathit{Init}(t^\dagger)$, which relates the starting points of an evaluation and an execution. We require the graph $E^\ddagger \circ t^\dagger$ to have no edges between links, which is based on the discussion at the end of Sec. 2.3 and essential for time-cost analysis. Although the definition of the translations and the operation $\circ$ on graphs use edges between links (e.g. the translation $x^\dagger$), such edges can be eliminated as soon as they are introduced, by identifying endpoints. For example, a variable can be translated into a single link that is both an input and an output, and outputs of the translation $(t \mathbin{@} u)^\dagger$ can be simply the union of outputs of $t^\dagger$ and $u^\dagger$. The graph $E^\ddagger \circ t^\dagger$ can be constructed by identifying interfaces of $E^\ddagger$ and $t^\dagger$, instead of introducing edges.

The binary relation $\preceq$ gives a weak simulation of the sub-machine semantics by the graph-rewriting machine. The weakness, i.e. the extra transitions compared with reductions, comes from the locality of pass transitions and the bureaucracy of managing !-boxes.

**Theorem 2.4.4** (Weak simulation with global bound)**.**

1. *If* $E\langle(\!|t|\!)\rangle \multimap_\chi E'\langle(\!|t'|\!)\rangle$ *and* $E\langle(\!|t|\!)\rangle \preceq ((E^\ddagger \circ t^\dagger, e), \delta)$ *hold, then there exists a number* $n \leq 3$ *and a graph state* $(((E')^\ddagger \circ (t')^\dagger, e'), \delta')$ *such that* $((E^\ddagger \circ t^\dagger, e), \delta) \to_\epsilon^n \to_\chi (((E')^\ddagger \circ (t')^\dagger, e'), \delta')$ *and* $E'\langle(\!|t'|\!)\rangle \preceq (((E')^\ddagger \circ (t')^\dagger, e'), \delta')$.

2. *If* $A\langle(\!|v|\!)\rangle \preceq ((A^\ddagger \circ v^\dagger, e), \delta)$ *holds, then the graph state* $((A^\ddagger \circ v^\dagger, e), \delta)$ *is initial, from which only the transition* $\mathit{Init}(A^\ddagger \circ v^\dagger) \to_\epsilon \mathit{Final}(A^\ddagger \circ v^\dagger)$ *is possible.*

*Proof.* For the second half, $e$ is the root of the graph $A^\ddagger \circ v^\dagger$, which means the state $((A^\ddagger \circ v^\dagger, e), \delta)$ is not a result of any pass transition. Therefore, by the condition (ii) of the binary relation $\preceq$, we have $\mathit{Init}(A^\ddagger \circ v^\dagger) = ((A^\ddagger \circ v^\dagger, e), \delta)$, and one pass transition from this state yields a final state $\mathit{Final}(A^\ddagger \circ v^\dagger)$.

For the first half, Fig. 2.14, Fig. 2.15 and Fig. 2.16 illustrate how the graph-rewriting machine simulates each reduction $\multimap$ of the sub-machine semantics. Each sequence of transitions $\to$ simulates a single reduction $\multimap$. Annotations of edges are omitted, and only the first and the last states of each sequence are shown, except for the case of the basic rule (2.10).

Some sequences involve equations that apply the four decomposition properties of the translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$, which are given earlier in this section. These equations rely on the fact that terms are alpha-converted in a form in which all binders introduce distinct variables, and reductions with labels $\beta$ and $\sigma$ work modulo alpha-equivalence to avoid name captures. This implies the following.

- Free variables of $u$ are not captured by $A$ in the case of the basic rule (2.2).
- Free variables of $A'\langle v\rangle$ are not captured by $A$ in the case of the basic rules (2.5) and (2.8).
- The variable $x$ is not captured by $E$ or $E'$ in the case of the basic rules (2.9) and (2.10).
- In the case of the basic rule (2.10), free variables of $E'$ are not captured by $A$, free variables of $v$ are not captured by $E'$, and $x$ does not freely appear in $v$.

Simulation of the basic rule (2.10) involves duplicating the sub-graph $v^\dagger$, which is a !-box. Because free variables of the value $v$ are captured by either $E$ or $A$, the multiset $\mathrm{FV}(v)$ can be partitioned into two multisets as $\mathrm{FV}(v) = M_E + M_A$, such that $M_E$ is the multiset of those captured by $E$ and $M_A$ is the multiset of those captured by $A$. No variable is contained by both $M_E$ and $M_A$. The translations $E^\ddagger$ and $A^\dagger$ include $C$-nodes that correspond to $M_E$ and $M_A$, respectively. These $C$-nodes get extra inputs by the rewrite transition labelled with $\sigma$, as represented by the middle state in the simulation sequence.

In each sequence, let $G_s$ and $G_t$ be the first and the last graph, respectively. By the condition (ii) of the binary relation $\preceq$, there exists an execution *Exec* :

$\mathit{Init}(G_s) \rightarrow^* ((G_s, e_1), (\uparrow, \square, S', B'))$ of only pass transitions, in which the link $e_1$ (see the figures) appears as a token position only once at the end.

1. In simulation of the basic rules (2.1), (2.3) and (2.6), the figures use $S$ and $B$ instead of $S'$ and $B'$. By Lem. 2.3.3, the result position $e_2$ (see the figures) does not appear in the execution *Exec*; if this is not the case, $e_1$ would appear more than once in *Exec*, which is a contradiction. Therefore, *Exec* followed by the pass transitions shown in the figures gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

2. In simulation of the basic rule (2.9), the figure uses $S$ and $B$ instead of $S'$ and $B'$. Because $x$ is not captured by $E'$, the starting position $e_1$ is in fact an input of the $C_{m+1}$-node. Using Lem. 2.3.3 again in the same way, the result position $e_2$ does not appear in the execution *Exec*. Therefore, *Exec* followed by the pass transition shown in the figures gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

3. In simulation of the basic rule (2.7), by the reversibility of pass transitions, there exist stacks $S$ and $B$ such that: $S' = S$, $B' = \star : B$, and the execution *Exec* can be decomposed into an execution $\mathit{Exec}' : \mathit{Init}(G_s) \rightarrow^* ((G_s, e_0), (\uparrow, \square, S, B))$ and one subsequent pass transition (see the figure for $e_0$). In the execution $\mathit{Exec}'$, the link $e_0$ appears as a token position only once at the end, which can be checked by contradiction as follows.

   - If $e_0$ appears more than once in $\mathit{Exec}'$ and its first appearance is with direction $\downarrow$, it must be a result of a pass transition. However, no pass transition leads to this situation, because $e_0$ is an input of a function application node. This is a contradiction.
   - If $e_0$ appears more than once in $\mathit{Exec}'$ and its first appearance is with direction $\uparrow$, it must be with rewrite flag $\square$, because $\mathit{Exec}'$ consists of pass

transitions only. Regardless of token data, the first appearance leads to an extra appearance of $e_1$ in $Exec'$, which is a contradiction.

Given this freshness of $e_0$ in $Exec'$, by Lem. 2.3.3, the result position $e_2$ does not appear in the execution $Exec'$. Therefore, $Exec$ followed by the pass transitions shown in the figures gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

4. In simulation of the basic rules (2.2), (2.5) and (2.8), by the reversibility of pass transitions, there exist stacks $S$ and $B$ such thatthe execution $Exec$ can be decomposed into an execution $Exec' : Init(G_s) \rightarrow^* ((G_s, e_0), (\uparrow, \square, S, B))$ and at least one subsequent pass transition. In the execution $Exec'$, the link $e_0$ appears as a token position only once at the end, which can be checked in the same manner as the previous case (3). Using this freshness of $e_0$ in $Exec'$ and Lem. 2.3.3, we can conclude that any node that interacts with a token in the execution $Exec'$ (i.e. that is relevant in a pass transition in the execution $Exec'$) belongs to $E^{\ddagger}$. This means that any pass transition in $Exec'$, on the starting graph $G_s$, can be imitated in the resulting graph $G_t$. Namely, the link $e_0$ corresponds to the result position $e_2$, and $Exec'$ corresponds to an execution $Exec'' : Init(G_t) \rightarrow^* ((G_t, e_2), (\uparrow, \square, S, B))$ of only pass transitions, in which $e_2$ appears only once at the end. This execution $Exec''$ gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

5. In simulation of the basic rule (2.4), the same reasoning as the previous case (4) gives an execution $Exec'' : Init(G_t) \rightarrow^* ((G_t, e_0), (\uparrow, \square, S, B))$ of only pass transitions, in which $e_0$ appears only once at the end. By Lem. 2.3.3, the result position $e_2$ does not appear in the execution $Exec''$. Therefore, $Exec''$ followed by pass transitions gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

6. In simulation of the basic rule (2.10), by the reversibility of pass transitions,

there exist an input $e_0$ of the $C_{m+1}$-node and stacks $S$ and $B$ such that: $S' = S$, $B' = e_0 : B$, and the execution *Exec* can be decomposed into an execution $Exec' : Init(G_s) \rightarrow^* ((G_s, e_0), (\uparrow, \square, S, B))$ and one subsequent pass transition that pushes $e_0$ to the box stack. By Lem. 2.3.3, the link $e_3$ (see the figure) appears in the execution $Exec'$. Analysing this appearance, we can conclude that the link $e_0$ is in fact the main output of $(E')^{\ddagger}_{\emptyset}$.

- If $e_3$ appears with direction $\downarrow$ in $Exec'$, because $e_3$ is an input of a function application node or a $C$-node, this appearance cannot be a result of any pass transition. This is a contradiction.
- If $e_3$ appears with direction $\uparrow$, it must be with rewrite flag $\square$, because $Exec'$ consists of pass transitions only. Because $e_3$ is the main input of $(E')^{\ddagger}_{\emptyset}$, by Lem. 2.4.2, this appearance leads to a state whose token position is the main output $e'$ of $(E')^{\ddagger}_{\emptyset}$, direction is $\uparrow$ and rewrite flag is $\square$. One pass transition from the state leads to a state whose token position is $e_1$. This means there exists an execution $Exec'''$ of pass transitions only, via the token position $e_3$ and the second last token position $e'$, to the token position $e_1$. Because pass transitions are deterministic, it is either: (1) *Exec* is strictly a sub-sequence of $Exec'''$, (2) $Exec = Exec'''$, or (3) $Exec'''$ is strictly a sub-sequence of *Exec*. Because *Exec* is followed by a pass transition and a rewrite transition as shown in the figure, the case (1) is impossible. Because $e_1$ appears only once at the end in the execution *Exec*, the case (3) leads to a contradiction. Therefore we can conclude that (2) is the case, i.e. $Exec = Exec'''$. This means $e' = e_0$, i.e. $e_0$ is the main output of $(E')^{\ddagger}_{\emptyset}$.

As a consequence, the link $e_2$ is indeed the result position, corresponding to the link $e_0$.

The rest of the reasoning is similar to the case 4. In the execution *Exec* to

the starting position $e_1$, the token does not interact with nodes that belong to $A^\dagger$ or $v^\dagger$; otherwise, by Lem. 2.3.3, $e_1$ would have an extra appearance in *Exec*, which is a contradiction. For the same reason, the execution *Exec*$'$ to the link $e_0$ does not involve any interaction of the token with the $C_{m+1}$-node, and hence $e_0$ appears only once at the end in the execution *Exec*$'$. As a result, the execution *Exec*$'$ gives an execution $\mathit{Exec}'' : \mathit{Init}(G_t) \to^* ((G_t, e_2), (\uparrow, \square, S, B))$ of only pass transitions on the resulting graph $G_t$, in which $e_2$ appears only once at the end. This execution *Exec*$''$ gives a desired execution that meets the condition (ii) of the binary relation $\preceq$.

□

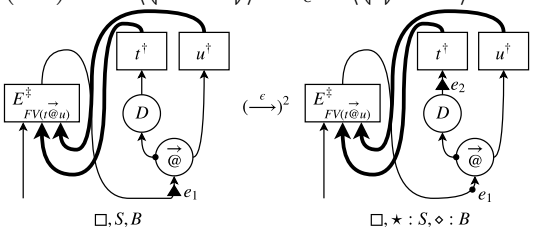
## 2.5 Time-cost analysis

We analyse how time-efficiently the token-guided graph-rewriting machine implements evaluation strategies, following the methodology developed by Accattoli et al. [2014], Accattoli and Sacerdoti Coen [2014], Accattoli [2017]. The time-cost analysis focuses on how efficiently an abstract machine implements an evaluation strategy. In other words, we are not interested in efficiency of evaluation strategies themselves, nor in minimising the number of $\beta$-reduction steps simulated by an abstract machine. Our aim is to see if the number of transitions of an abstract machine is *reasonable*, compared to the number of necessary $\beta$-reduction steps determined by a given evaluation strategy.

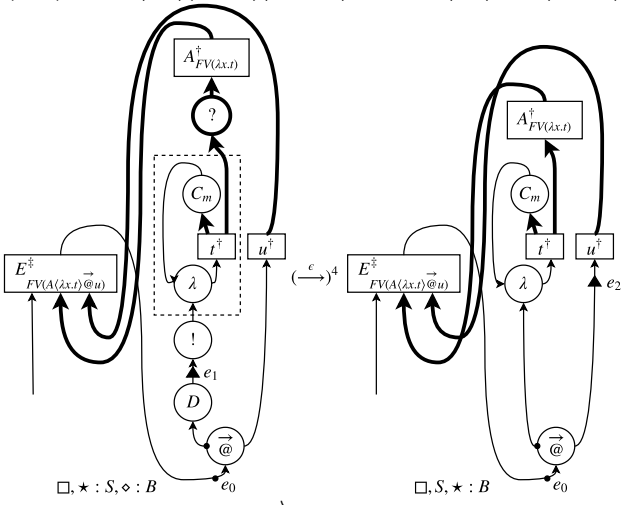
Accattoli's methodology assumes that an abstract machine has three groups of transitions: 1) *$\beta$-transitions* that correspond to $\beta$-reduction in which substitution is delayed, 2) transitions that perform substitution, and 3) other *overhead* transitions. We incorporate this classification using the labels $\beta$, $\sigma$ and $\epsilon$ of transitions.

Another assumption of the methodology is that, a single transition of an abstract machine is enough to simulate each step of $\beta$-reduction, or each step of substitu-

(2.3) $E\langle (\!|t \overrightarrow{@} u|\!)\rangle \multimap_\epsilon E\langle (\!|t|\!) \overrightarrow{@} u\rangle$




(2.4) $E\langle A\langle (\!|\lambda x.t|\!)\rangle \overrightarrow{@} u\rangle \multimap_\epsilon E\langle A\langle \lambda x.t\rangle \overrightarrow{@} (\!|u|\!)\rangle$




(2.5) $E\langle A\langle \lambda x.t\rangle \overrightarrow{@} A'\langle (\!|v|\!)\rangle\rangle \multimap_\beta E\langle A\langle (\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle\rangle$

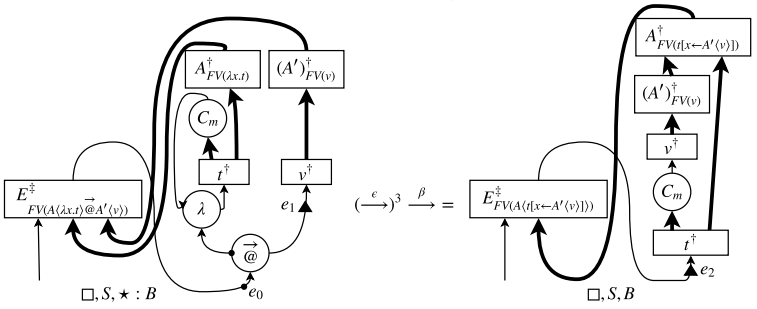
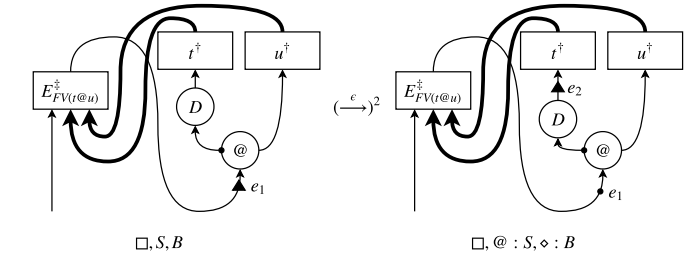
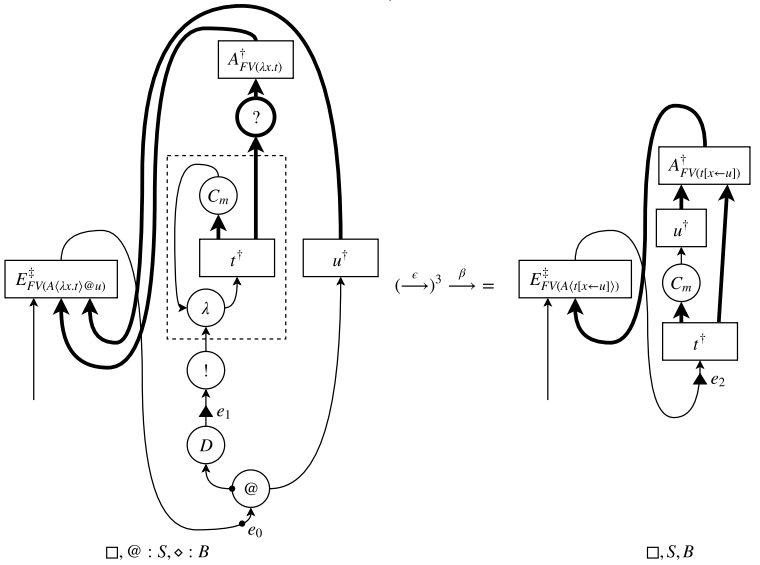



Figure 2.14: Illustration of simulation: left-to-right call-by-value application

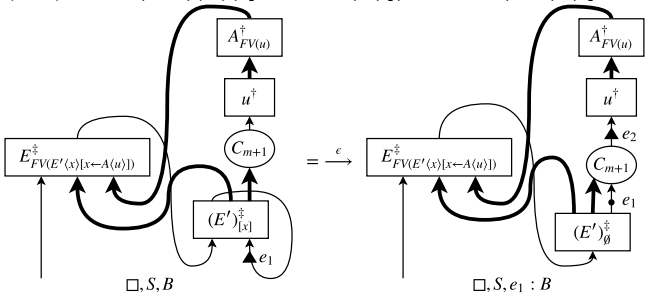
(2.1) $E\langle (\!| t \mathbin{@} u |\!) \rangle \multimap_{\epsilon} E\langle (\!| t |\!) \mathbin{@} u \rangle$




(2.2) $E\langle A\langle (\!| \lambda x.t |\!) \rangle \mathbin{@} u \rangle \multimap_{\beta} E\langle A\langle (\!| t |\!) [x \leftarrow u] \rangle \rangle$




(2.9) $E\langle E'\langle (\!| x |\!) \rangle [x \leftarrow A\langle u \rangle] \rangle \multimap_{\epsilon} E\langle E'\langle x \rangle [x \leftarrow A\langle (\!| u |\!) \rangle] \rangle$




(2.10) $E\langle E'\langle x \rangle [x \leftarrow A\langle (\!| v |\!) \rangle] \rangle \multimap_{\sigma} E\langle A\langle E'\langle (\!| v |\!) \rangle [x \leftarrow v] \rangle \rangle$

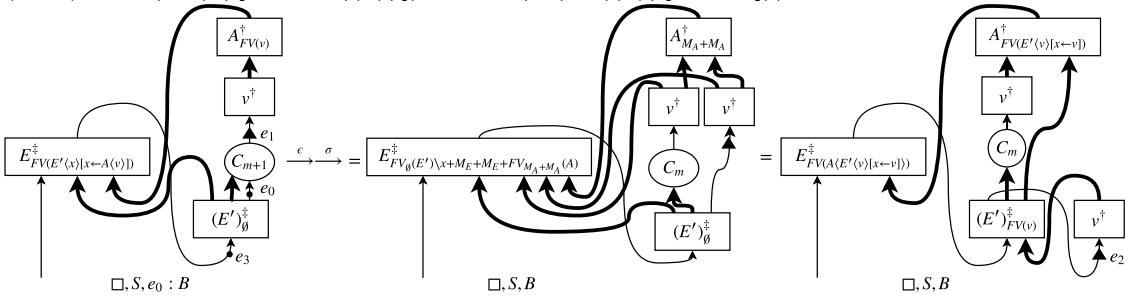
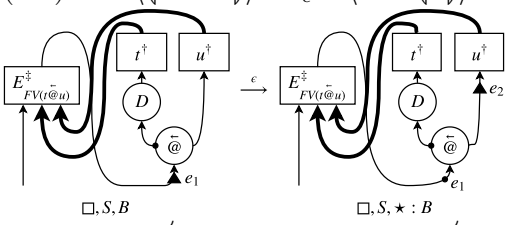
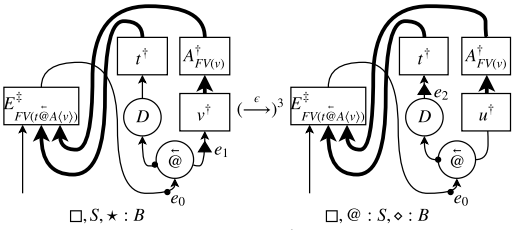



Figure 2.15: Illustration of simulation: call-by-need application and explicit substitutions

(2.6) $E\langle (\!| t \overleftarrow{@} u |\!) \rangle \multimap_\epsilon E\langle t \overleftarrow{@} (\!| u |\!) \rangle$




(2.7) $E\langle t \overleftarrow{@} A\langle (\!| v |\!) \rangle \rangle \multimap_\epsilon E\langle (\!| t |\!) \overleftarrow{@} A\langle v \rangle \rangle$




(2.8) $E\langle A\langle (\!| \lambda x.t |\!) \rangle \overleftarrow{@} A'\langle v \rangle \rangle \multimap_\beta E\langle A\langle (\!| t |\!) [x \leftarrow A'\langle v \rangle] \rangle \rangle$

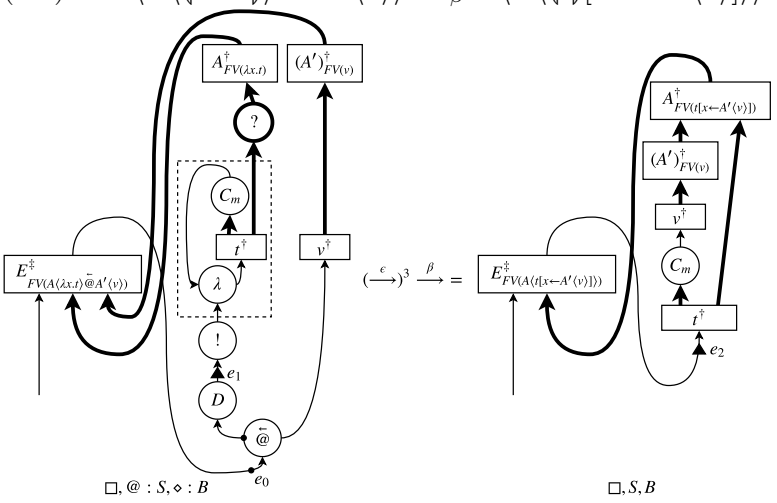



Figure 2.16: Illustration of simulation: right-to-left call-by-value application

tion in which one occurrence of a variable is replaced. This is satisfied by many known abstract machines, including the storeless abstract machine of Danvy and Zerny [2013] which our sub-machine semantics resembles, however not by the token-guided graph-rewriting abstract machine. The machine has finer transitions and can take several transitions to simulate a single step of reduction, as we can observe in Thm. 2.4.4. In spite of this mismatch we can still follow the methodology, thanks to the weak simulation $\preceq$. It discloses what transitions of the token-guided graph-rewriting machine exactly correspond to $\beta$-reduction and substitution, and gives a concrete number of overhead transitions that the machine needs to simulate

$\beta$-reduction and substitution.

The methodology of time-cost analysis has four steps: (I) bound the number of transitions required in implementing evaluation strategies, (II) estimate time cost of each transition, (III) bound overall time cost of implementing evaluation strategies, by multiplying the number of transitions with time cost for each transition, and finally (IV) classify the abstract machine according to its execution time cost. Consider now the following taxonomy of abstract machines introduced in Accattoli [2017].

**Definition 2.5.1** (classes of abstract machines [Accattoli, 2017, Def. 7.1])**.**

1. An abstract machine is said to be *efficient* if its execution time cost is linear in both the input size and the number of $\beta$-transitions.

2. An abstract machine is said to be *reasonable* if its execution time cost is polynomial in the input size and the number of $\beta$-transitions.

3. An abstract machine is said to be *unreasonable* if it is not reasonable.

In our case, the input size is given by the *size* $|t|$ of the term $t$, inductively defined by:

$$\begin{aligned} |x| &:= 1, \\ |\lambda x.t| &:= |t| + 1, \\ |t \mathbin{@} u| = |t \overrightarrow{@} u| = |t \overleftarrow{@} u| &:= |t| + |u| + 1, \\ |t[x \leftarrow u]| &:= |t| + |u| + 1. \end{aligned}$$

The number of $\beta$-transitions of the DGoIM is simply the number of transitions $\rightarrow_\beta$ labelled with $\beta$. Thm. 2.4.4 implies that this number in fact corresponds to the number of reductions $\multimap_\beta$, labelled with $\beta$, of the sub-machine semantics.

Given an evaluation *Eval*, the number of occurrences of a label $\chi$ is denoted by $|Eval|_\chi$. The sub-machine semantics comes with the following quantitative bounds.

**Proposition 2.5.2.** *For any pure closed term $t$ and any evaluation $Eval\colon (\!|t|\!) \multimap^* A\langle (\!|v|\!)\rangle$ that terminates, the number of reductions is bounded by $|Eval|_\sigma = \mathcal{O}(|Eval|_\beta)$ and $|Eval|_\epsilon = \mathcal{O}(|t| \cdot |Eval|_\beta)$.*

*Proof.* The first equation can be proved by directly applying the methodology of time-cost analysis by Accattoli et al. [2014] to the sub-machine semantics. Its outline is as follows.

Recall our assumption that each term involves function applications of a single evaluation strategy only, which is either one of these three: call-by-need, left-to-right call-by-value, or right-to-left call-by-value. Using the concept of *distillery* [Accattoli et al., 2014, Sec. 4], we can establish simulation between evaluations by the sub-machine semantics and *derivations* in three linear substitution calculi [Accattoli and Kesner, 2010], each of which uses one of the three evaluation strategies. At the core of the simulation is the fact that enriched terms of the sub-machine semantics can be turned into terms of the linear substitution calculi by simply forgetting the windows.

The simulation in particular entails that each step of reduction $\multimap_\beta$ (resp. $\multimap_\sigma$) of the sub-machine semantics is simulated by exactly one *multiplicative* (resp. *exponential*) derivation step in the linear substitution calculi. This enables us to use the complexity result about the three linear substitution calculi [Accattoli and Sacerdoti Coen, 2014, Cor. 1 & Thm. 2], which states that, in derivations that successfully terminate, the number of exponential steps is linear with respect to the number of multiplicative steps. Consequently, the number of reductions $\multimap_\sigma$ is linear with respect to the number of reductions $\multimap_\beta$ in evaluations that successfully terminate, and this is exactly the first equation $|Eval|_\sigma = \mathcal{O}(|Eval|_\beta)$.

The second equation is proved by combining the first equation and an equation $|Eval|_\epsilon = \mathcal{O}(|t| \cdot (|Eval|_\beta + |Eval|_\sigma))$. This auxiliary equation can be proved using ideas from analysis of various abstract machines by Accattoli et al. [2014, Thm. 11.3 & Thm. 11.5], as below.

For any enriched term $E'\langle(\!|t'|\!)\rangle$ that appears in the evaluation $Eval\colon (\!|t|\!) \multimap^* A\langle(\!|v|\!)\rangle$, we define two measures. The first measure $\#_1(E'\langle(\!|t'|\!)\rangle)$ is defined by: $|t'|+|u|$ if $E'$ is in the form of $E''\langle A'\langle\cdot\rangle @ u\rangle$, $E''\langle A'\langle\cdot\rangle \overrightarrow{@}\, u\rangle$, or $E''\langle u \overleftarrow{@}\, A'\langle\cdot\rangle\rangle$; and $|t'|$ otherwise. By Lem. 2.2.1, both $t'$ and $u$ above are sub-terms of $t$, and we have $\#_1(E'\langle(\!|t'|\!)\rangle) \leq 2\cdot|t|$. The second measure $\#_2(E')$ is on $E'$ only, and defined inductively as below.

$$\begin{aligned}
\#_2(\langle\cdot\rangle) &:= 0,\\
\#_2(E''\langle x\rangle[x \leftarrow E''']) &:= \#_2(E'') + \#_2(E''') + 1,\\
\#_2(E'' \mathbin{@} t) = \#_2(E'' \overrightarrow{@}\, t) = \#_2(A\langle v\rangle \overrightarrow{@}\, E'') &:= \#_2(E''),\\
\#_2(t \overleftarrow{@}\, E'') = \#_2(E'' \overleftarrow{@}\, A\langle v\rangle) = \#_2(E''[x \leftarrow t'']) &:= \#_2(E'').
\end{aligned}$$

Because the basic rules (2.1), (2.3), (2.4), (2.6) and (2.7) strictly reduce the measure $\#_1$, these rules can be consecutively applied at most $2\cdot|t|$ times. The evaluation *Eval* can be seen as applications of these rules interleaved with other rules, so the total number of applications of these five basic rules can be bounded by $\mathcal{O}(|t|\cdot(|Eval|_\beta + |Eval|_\sigma + |Eval|_9))$, where $|Eval|_9$ denotes the total number of applications of the basic rule (2.9).

The measure $\#_2$ is increased only by the basic rule (2.9) and decreased only by the basic rule (2.10). Both the increase and the decrease are of one. Because the measure $\#_2$ gives zero for both $(\!|t|\!)$ and $A\langle(\!|v|\!)\rangle$, namely $\#_2(\langle\cdot\rangle) = \#_2(A) = 0$, the basic rule (2.9) must be applied as many times as the basic rule (2.10) in the evaluation *Eval*. This means $|Eval|_\sigma = |Eval|_9$.

Combining the bound $\mathcal{O}(|t|\cdot(|Eval|_\beta + |Eval|_\sigma + |Eval|_9))$ with the equation $|Eval|_\sigma = |Eval|_9$ gives the auxiliary equation on $|Eval|_\epsilon$. □

We use the same notation $|Exec|_\chi$, as for an evaluation, to denote the number of occurrences of each label $\chi$ in an execution *Exec*. Additionally the number of rewrite transitions with the label $\epsilon$, i.e. those that eliminates a !-box structure, is denoted by $|Exec|_{\epsilon R}$. Note that pass transitions are all labelled with $\epsilon$, and hence

$|Exec|_{\epsilon R} \leq |Exec|_{\epsilon}$. The following proposition completes the first step of the cost analysis.

**Proposition 2.5.3** (Soundness and completeness, with transition bounds)**.** *For any pure closed term $t$, an evaluation $Eval\colon (\!|t|\!) \multimap^* A\langle (\!|v|\!)\rangle$ terminates with the enriched term $A\langle (\!|v|\!)\rangle$ if and only if an execution $Exec\colon Init(t^\dagger) \to^* Final(A^\ddagger \circ v^\dagger)$ terminates with the graph $A^\ddagger \circ v^\dagger$. Moreover the number of transitions is bounded by $|Exec|_\beta = |Eval|_\beta$, $|Exec|_\sigma = \mathcal{O}(|Eval|_\beta)$, $|Exec|_\epsilon = \mathcal{O}(|t| \cdot |Eval|_\beta)$, $|Exec|_{\epsilon R} = \mathcal{O}(|Eval|_\beta)$.*

*Proof.* Because the initial term $t$ is closed, any enriched term $E'\langle (\!|t'|\!)\rangle$ that appears in the evaluation *Eval* is also closed. This implies that a reduction is always possible at $E'\langle (\!|t'|\!)\rangle$ unless it is in the form of $A'\langle v'\rangle$. In particular, if $t'$ is a variable, the variable is captured by an explicit substitution in $E'$ and the basic rule (2.10) is possible. Consequently, if an evaluation of the pure closed term $t$ terminates, the last enriched term is in the form of $A'\langle v'\rangle$.

The forward direction of the equivalence, that is, the evaluation *Eval* implies the execution *Exec*, follows from Thm. 2.4.4. The backward direction, that is, the execution *Exec* implies the evaluation *Eval*, also follows from Thm. 2.4.4, because an evaluation of the pure closed term $t$ is in the form of $(\!|t|\!) \multimap^* A\langle (\!|v|\!)\rangle$ or never terminates.

Thm. 2.4.4 also gives equations $|Exec|_\beta = |Eval|_\beta$, $|Exec|_\sigma = |Eval|_\sigma$ and $|Exec|_\epsilon = \mathcal{O}(|Eval|_\beta + |Eval|_\sigma + |Eval|_\epsilon)$. Combining these with Prop. 2.5.2 yields the desired equations except for the last one (i.e. $|Exec|_{\epsilon R} = \mathcal{O}(|Eval|_\beta)$).

This last equation follows from an equation $|Exec|_{\epsilon R} = |Exec|_\beta$ that can be proved as follows. For any graph state $((G, e), \delta)$ that appears in the execution $Exec\colon Init(t^\dagger) \to^* Final(A^\ddagger \circ v^\dagger)$, we define a measure $\#(G)$ by the number of $\lambda$-nodes that are outside any !-box in the graph $G$.

Firstly, at any point of the execution *Exec*, the token is inside a !-box if and only if it has the rewrite flag '!'. This means that, if a $\lambda$-node gets eliminated by a rewrite transition labelled with $\beta$, the $\lambda$-node is outside a !-box. By Lem. 2.3.4,

each !-box has exactly one $\lambda$-node that directly belongs to it. It follows that each rewrite transition labelled with $\epsilon$ brings exactly one $\lambda$-node outside a !-box.

As a result, each rewrite transition labelled with $\beta$ decreases the measure # by one, and each rewrite transition labelled with $\epsilon$ increases the measure # by one. No other transitions change the measure #. Because the measure # gives zero for the initial and final graph states $\mathit{Init}(t^\dagger)$ and $\mathit{Final}(A^\ddagger \circ v^\dagger)$, namely $\#(t^\dagger) = \#(A^\ddagger \circ v^\dagger) = 0$, we have $|\mathit{Exec}|_{\epsilon R} = |\mathit{Exec}|_\beta$. □

The next step in the cost analysis is to estimate the time cost of each transition. We are interested in implementing evaluation strategies, and therefore we focus on transitions that happen in executions starting from the translation of a term. We assume that graphs are implemented in the following particular way, for the purposes that we will explain shortly afterwards.

Each ?-node, and its input and output, are identified and implemented as a single link. Each link is given by two pointers to its child and its parent. If a node is not a ?-node, it is given by its label, pointers to its inputs, and pointers to its outputs; the pointers to inputs are omitted for $C$-nodes. Additionally, each link and node has a pointer to a !-node, or a null pointer, to indicate the !-box structure it directly belongs in. Note that each link has at most three pointers, and each node has at most two input (resp. output) pointers, which are distinguished. The *size* of a graph can be estimated using the number of nodes that are not ?-nodes. Accordingly, a position of the token is a pointer to a link, a direction and a rewrite flag are two symbols, a computation stack is a stack of symbols, and finally a box stack is a stack of symbols and pointers to links.

There are two key purposes of these rather involved assumptions of implementation. One purpose is to bound the number of pointers that represent each node. Pointers to inputs are omitted at $C$-nodes for this purpose, because these nodes are the only ones that can have an arbitrary number of inputs. The other purpose is to estimate that the translation of terms yields linear representation, namely that the

translation $t^\dagger$ of each term $t$ has the size that is linear to the size $|t|$ of the term. This estimation is impossible without the assumption that each !-box structure is implemented using pointers to its principal door (!-node) and omitting auxiliary doors (?-nodes). Without this assumption, a single variable may be translated using multiple ?-nodes whose number can only be bounded by the size of the term, which leads to not linear but polynomial representation.

The assumption about implementation of !-boxes is also for the purpose of determining !-boxes simply by traversing nodes, in executions that start from the translation of a term. At any point of these executions, each !-box already appeared as a sub-graph of the initial graph (by Lem. 2.3.4), which is the translation of a term. This means that the !-box is always the translation of abstraction, and moreover, every node inside the !-box is reachable from the principal door (!-node) of the !-box. The !-box structure can therefore be recovered by traversing nodes from the principal door. The end of traversal can be determined using the assumed pointers from nodes to !-nodes, and the traversal cost can be bounded by the size of the !-box.

Under the assumptions about implementation, time cost of each transition can be finally estimated as follows. All pass transitions have constant cost. Each pass transition looks up one node and its outputs (that are either one or two) next to the current position, and involves a fixed number of elements of the token data. Rewrite transitions with the label $\beta$ have constant cost, as they change a constant number of nodes and links, and only a rewrite flag of the token data.

Rewrite transitions with the label $\epsilon$ or $\sigma$ manipulate !-boxes, namely, those with the label $\epsilon$ remove a !-box and those with the label $\sigma$ copy a !-box. Both these manipulations amount to traversing nodes in the !-box, whose cost can be bounded by the size of the !-box. Additionally, rewrite transitions with the label $\sigma$ update the sub-graph $H'$ and a $C$-node connected to the copied !-box (see Fig. 2.6). Updating cost of $H'$ is bounded by the number of auxiliary doors of the !-box, which is less than the size of the !-box. Updating cost of the $C$-node is constant, because $C$-nodes

do not have pointers to its inputs, by the assumption about the implementation of graphs. Overall, rewrite transitions with the label $\epsilon$ or $\sigma$ have the time cost bounded by the size of the involved !-box.

With the results of the previous two steps, we can now give the overall time cost of executions and classify our abstract machine.

**Theorem 2.5.4** (Soundness and completeness, with cost bounds)**.** *For any pure closed term $t$, an evaluation $Eval\colon (\!|t|\!) \multimap^* A\langle (\!|v|\!)\rangle$ terminates with the enriched term $A\langle (\!|v|\!)\rangle$ if and only if an execution $Exec\colon Init(t^\dagger) \to^* Final(A^\ddagger \circ v^\dagger)$ terminates with the graph $A^\ddagger \circ v^\dagger$. The overall time cost of the execution $Exec$ is bounded by $\mathcal{O}(|t| \cdot |Eval|_\beta)$.*

*Proof.* Non-constant cost of rewrite transitions is the size of a !-box. By Lem. 2.3.4, this size is less than the size of the initial graph $t^\dagger$, which can be bounded by the size $|t|$ of the initial term. Therefore any non-constant cost of each rewrite transition, in the execution $Exec$, can be also bounded by $|t|$. By Prop. 2.5.3, the overall time cost of rewrite transitions labelled with $\beta$ is $\mathcal{O}(|Eval|_\beta)$, and that of the other rewrite transitions and pass transitions is $\mathcal{O}(|t| \cdot |Eval|_\beta)$. □

Note that the time cost of constructing the initial graph $t^\dagger$, and attaching a token to it, does not affect the bound $\mathcal{O}(|t| \cdot |Eval|_\beta)$, because it can be done in linear time with respect to $|t|$. This is thanks to the assumption about implementation, namely that ?-nodes and input pointers of $C$-nodes are omitted.

**Corollary 2.5.5.** *The token-guided graph-rewriting machine is an efficient abstract machine implementing call-by-need, left-to-right call-by-value and right-to-left call-by-value evaluation strategies, in the sense of Def. 2.5.1.*

Cor. 2.5.5 classifies the graph-rewriting machine as not just *reasonable*, but in fact *efficient*. In terms of token passing, this efficiency benefits from the graphical representation of environments (i.e. explicit substitutions in our setting). The

graphical representation is in such a way that each bound variable is associated with exactly one $C$-node, which is ensured by the translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$ and the rewrite transition $\rightarrow_\sigma$. Excluding any two sequentially-connected $C$-nodes is essential to achieve the efficient classification, because it yields the constant cost to look up a bound variable and its associated computation.

As for graph rewriting, the efficient classification shows that introduction of graph rewriting to token passing does not bring in any inefficiencies. In our setting, graph rewriting can have non-constant cost in two ways. One is duplication cost of a sub-graph, which is indicated by a !-box, and the other is elimination cost of a !-box that delimits abstraction. Unlike the duplication cost, the elimination cost leads to non-trivial cost that abstract machines in the literature usually do not have. Namely, our graph-rewriting machine simulates a $\beta$-reduction step, in which an abstraction constructor is eliminated and substitution is delayed, at the non-constant cost depending on the size of the abstraction. The time-cost analysis confirms that the duplication cost and the unusual elimination cost have the same impact, on the overall time cost, as the cost of token passing. What is vital here is the sub-graph property (Lem. 2.3.4), which ensures that the cost of each duplication and elimination of a !-box is always linear in the input size.

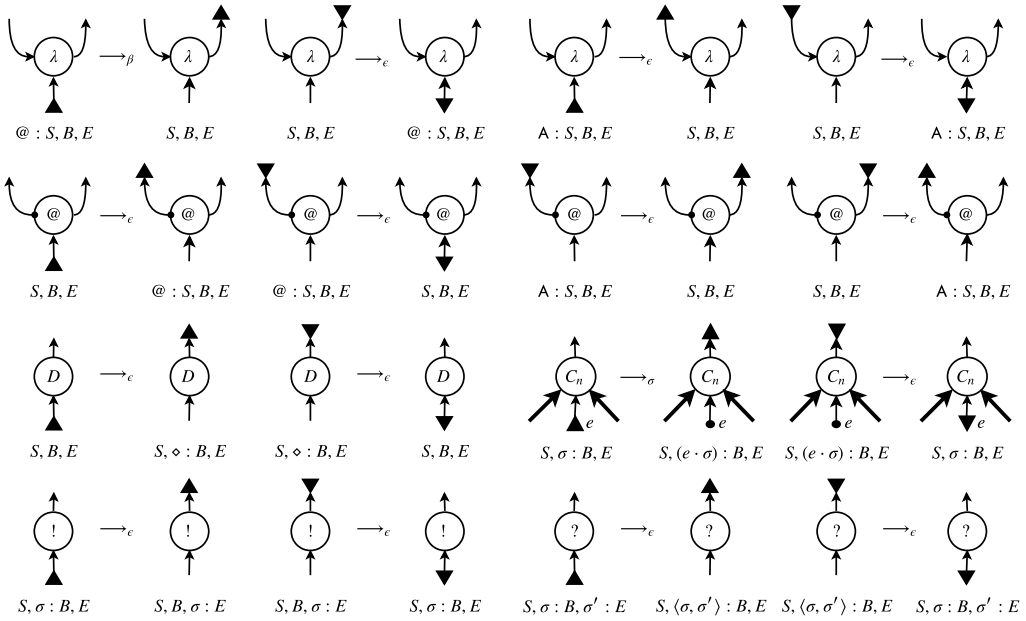
## 2.6 Rewriting vs. jumping

The starting point of our development is the GoI-style token-passing abstract machines for call-by-name evaluation, given by Danos and Regnier [1996], and by Mackie [1995]. Fig. 2.17 recalls these token-passing machines as a version of the DGoIM with the passes-only interleaving strategy (i.e. the DGoIM with only pass transitions). It follows the convention of Fig. 2.5, but a black triangle in the figure points along (resp. against) the direction of the edge if the token direction is $\uparrow$ (resp. $\downarrow$). Note that this version uses different token data, to which we will come back later.

Token data $(d, S, B, E)$ consists of:

- a *direction* defined by $d ::= \uparrow \mid \downarrow$,
- a *computation stack* defined by $S ::= \square \mid \mathsf{A} : S \mid @ : S$, and
- a *box stack* $B$ and an *environment stack* $E$, both defined by $B, E ::= \square \mid \sigma : B$, using *exponential signatures* $\sigma ::= \star \mid e \cdot \sigma \mid \langle \sigma, \sigma \rangle$ where $e$ is any link of the underlying graph.

Pass transitions:




Given a term $t$ with the call-by-need function application ($@$) abused, a successful execution $((t^\dagger, e_t), (\uparrow, \square, \square, \square, \square)) \to^* ((t^\dagger, e_v), (\uparrow, \square, \square, \square, \square))$ starts at the root $e_t$ of the translation $t^\dagger$, and ends at the root $e_v$ of the translation $v^\dagger$, for some sub-value $v$ of the term $t$. The value $v$ indicates the evaluation result.

Figure 2.17: Passes-only DGoIM for call-by-name [Danos and Regnier, 1996, Mackie, 1995]

Token-passing GoI keeps the underlying graph fixed, and re-evaluates a term by repeating token moves. It therefore favours space efficiency at the cost of time efficiency. The repetition of token actions poses a challenge for evaluations in which duplicated computation must not lead to repeated evaluation, especially call-by-value evaluation [Fernández and Mackie, 2002, Schöpp, 2014b, Hoshino et al., 2014, Dal Lago et al., 2015]. Moreover, in call-by-value the repetition of token actions raises the additional technical challenge of avoiding repeating any associated computational effects [Schöpp, 2011, Muroya et al., 2016, Dal Lago et al., 2017]. A partial solution to this conundrum is to focus on the soundness of the equational theory, while deliberately ignoring the time costs [Muroya et al., 2016]. Introduction of graph reduction, the key idea of the DGoIM, is one complete solution that also deals with the time costs. It namely avoids repeated token moves and also improves time efficiency of token-passing GoI. Another such solution in the literature is introduction of jumps. We discuss how these two solutions affect machine design and space efficiency.

The most greedy way of introducing graph reduction, namely the rewrites-first interleaving we studied in this work, simplifies machine design in terms of the variety of pass transitions and token data. First, some token moves turn irrelevant to an execution. This is why Fig. 2.5 for the rewrites-first interleaving has fewer pass transitions than Fig. 2.17 for the passes-only interleaving. Certain nodes, like '?', always get eliminated before visited by the token, in the rewrites-first interleaving. Accordingly, token data can be simplified. The box stack and the environment stack used in Fig. 2.17 are integrated to the single box stack used in Fig. 2.5. The integrated stack does not need to carry the exponential signatures. They make sure that the token exits !-boxes appropriately in the token-passing GoI, by maintaining binary tree structures, but the token never exits !-boxes with the rewrites-first interleaving. Although the rewrites-first interleaving simplifies token data, rewriting itself, especially duplication of sub-graphs, becomes the source of space-inefficiency.

A jumping mechanism can be added on top of the token-passing GoI, and enables the token to jump along the path it would otherwise follow step-by-step. Although no quantitative analysis is provided, it gives time-efficient implementations of evaluation strategies, namely of call-by-name evaluation [Danos and Regnier, 1996] and call-by-value evaluation [Fernández and Mackie, 2002]. Jumping can reduce the variety of pass transitions, like rewriting, by letting some nodes always be jumped over. Making a jump is just changing the token position, so jumping can be described as a variation of pass transitions, unlike rewriting. However, introduction of jumping rather complicates token data. Namely it requires partial duplication of token data, which not only complicates machine design but also damages space efficiency. The duplication effectively represent virtual copies of sub-graphs, and accumulate during an execution. Tracking virtual copies is the trade-off of keeping the underlying graph fixed. Some jumps that do not involve virtual copies can be described as a form of graph rewriting that eliminates nodes.

Finally, we give a quantitative comparison of space usage between rewriting and jumping. As a case study, we focus on implementations of call-by-name/need evaluation, namely on the passes-only DGoIM recalled in Fig. 2.17, our rewrites-first DGoIM, and the passes-only DGoIM equipped with jumping that we will recall in Fig. 2.18. A similar comparison is possible for left-to-right call-by-value evaluation, between our rewrites-first DGoIM and the jumping machine given by Fernández and Mackie [2002].

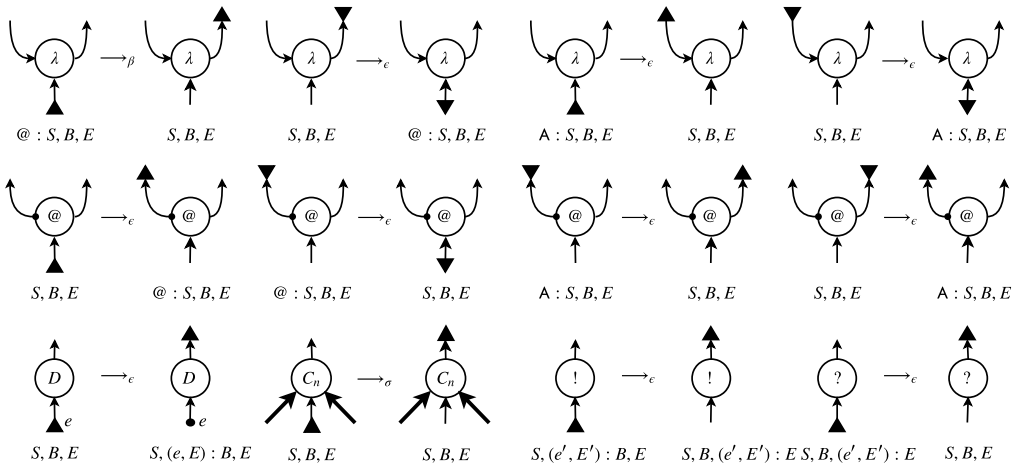
Fig. 2.18 recalls the token-passing machine equipped with jumping, given by Danos and Regnier [1996], which is proved to be isomorphic to Krivine's abstract machine [Krivine, 2007] for call-by-name evaluation. The machine has pass transitions as well as the *jump* transition that lets the token jump to a remote position.[7] Compared with the token-passing GoI (Fig. 2.17), pass transitions for nodes related to !-boxes are reduced and changed, so that the jumping mechanism imitates rewrites

[7]Our on-line visualiser additionally supports this jumping machine.

Token data $(d, S, B, E)$ consists of:

- a *direction* defined by $d ::= \uparrow \mid \downarrow$,
- a *computation stack* defined by $S ::= \square \mid \mathsf{A} : S \mid @ : S$, and
- a *box stack* $B$ and an *environment stack* $E$, both defined by $B, E ::= \square \mid (e, E) : B$, where $e$ is any link of the underlying graph.

Pass transitions:




Jump transition: $((G, e), (\downarrow, S, B, (e', E') : E)) \rightarrow_\epsilon ((G, e'), (\downarrow, S, B, E'))$, where the old position $e$ is the output of a !-node:  .

Given a term $t$ with the call-by-need function application (@) abused, a successful execution $((t^\dagger, e_t), (\uparrow, \square, \square, \square, \square)) \rightarrow^* ((t^\dagger, e_v), (\uparrow, \square, \square, \square, \square))$ starts at the root $e_t$ of the translation $t^\dagger$, and ends at the root $e_v$ of the translation $v^\dagger$, for some sub-value $v$ of the term $t$. The value $v$ indicates the evaluation result.

Figure 2.18: Passes-only DGoIM plus jumping for call-by-name [Danos and Regnier, 1996]

| machines | token-passing only (Fig. 2.17) | rewriting added (Fig. 2.5 & Fig. 2.6) | jumping added (Fig. 2.18) |
|---|---|---|---|
| evaluations implemented | call-by-name | call-by-need | call-by-name |
| size of graph | $\lvert G_0\rvert$ | $\mathcal{O}(n \cdot \lvert G_0\rvert)$ | $\lvert G_0\rvert$ |
| size of token position | $\log \lvert G_0\rvert$ | $\mathcal{O}(\log (n \cdot \lvert G_0\rvert))$ | $\log \lvert G_0\rvert$ |
| size of token data | $\mathcal{O}(n \cdot \log \lvert G_0\rvert)$ | $\mathcal{O}(n \cdot \log (n \cdot \lvert G_0\rvert))$ | $\mathcal{O}(2^n \cdot \log \lvert G_0\rvert)$ |

Table 2.1: Comparison between rewriting and jumping, case study: space usage after $n$ transitions from an initial state of a graph $G_0$

involving !-boxes. The token remembers its old position, together with its current environment stack, when passing a $D$-node upwards. The token uses this information and makes a jump back in the jump transition, in which the token exits a !-box at the principal door (!-node) and changes its position to the remembered link $e'$.

The quantitative comparison, whose result is stated below, shows that partial duplication of token data impacts space usage much more than duplication of sub-graphs, and therefore rewriting has asymptotically better space usage than jumping.

**Proposition 2.6.1.** *After $n$ transitions from an initial state of a graph of size $|G_0|$, space usage of three versions of the DGoIM is bounded as in Table 2.1.*

*Proof.* The size $|G_n|$ of the underlying graph after $n$ transitions can be estimated using the size $|G_0|$ of the initial graph. Our rewrites-first DGoIM is the only one that changes the underlying graph during an execution. Thanks to the sub-graph property (Lem. 2.3.4), the size $|G_n|$ can be bounded as $|G_n| = \mathcal{O}(n_\sigma \cdot |G_0|)$, where $n_\sigma$ is the number of $\sigma$-labelled transitions in the $n$ transitions. In the token-passing machines with and without jumping (Fig. 2.17 and Fig. 2.18), clearly $|G_n| = |G_0|$. In any of the three machines, the token position can be represented in the size of $\log |G_n|$.

Next estimation is of token data. Because stacks can have a link of the underlying graph as an element, the size of token data after $n$ transitions depends on $\log |G_n|$. Both in the token-passing machine (Fig. 2.17) and our rewrites-first DGoIM, at most

one element is pushed in each transition. Therefore the size of token data is bounded by $\mathcal{O}(n \cdot \log(|G_n|))$.

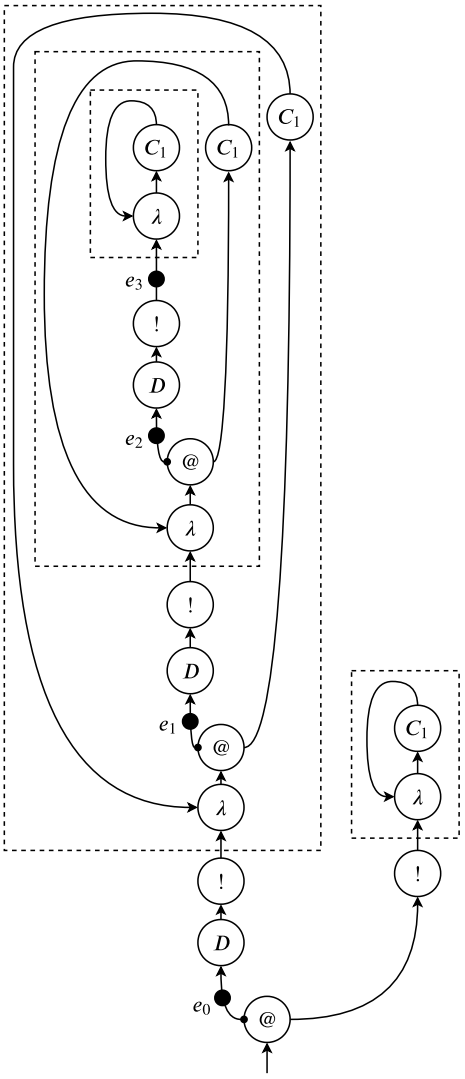
On the other hand, in the jumping machine (Fig. 2.18), it is only the computation stack that has at most linear growth during execution. The other stacks (i.e. the box stack and the environment stack) jointly grows at most exponentially, for the following reason.

Possible changes that each transition can make to these two stacks are: pushing a pair $(e, E)$ of a link $e$ and a copy of the environment stack $E$ onto the box stack; popping the top element of the environment stack; and simply moving the top element from the box stack to the environment stack. Only the first one among these changes increases the combined size of the box stack and the environment stack. Let $\#(S_n)$ and $\#(E_n)$ be the number of links stored in the box stack and the environment stack, respectively, after $n$ transitions. The combined number of links therefore satisfies $\#(S_{n+1})+\#(E_{n+1}) \leq \#(S_n)+2\cdot\#(E_n)+1 \leq 2\cdot(\#(S_n)+\#(E_n))+1$, which implies the exponential bound $\#(S_n) + \#(E_n) = \mathcal{O}(2^n)$.

Since the jumping machine never changes the underlying graph, the size of each link stored in stacks is always bounded by $\log(|G_0|)$. Consequently, the combined size of the box stack and the environment stack can be bounded by $\mathcal{O}(2^n \cdot \log(|G_0|))$. This overpowers the linear bound $\mathcal{O}(n)$ of the size of the computation stack, which stores symbols $\mathsf{A}$ and @. □

Fig. 2.19 illustrates an execution of the jumping machine where the token data, namely the environment stack, indeed grows exponentially before the token performs any jump. The execution is on a term $(\lambda x.(\lambda y.(\lambda a.a)y)\,x)\,(\lambda w.w)$. The term contains nested $\eta$-expansions, and these are translated into a specific pattern of $D$-nodes and !-nodes that causes the exponential growth. The translation of the term is shown in Fig. 2.19(a), where $e_0, e_1, e_2, e_3$ are links.

Fig. 2.19(b) shows how the box stack and the environment stack evolve, until the token reaches the link $e_3$ for the first time. Starting from the root of the graph,




(a) Translation of a term $(\lambda x.(\lambda y.(\lambda a.a)y)\,x)\,(\lambda w.w)$

| box stack | environment stack | position |
|---|---|---|
| $\square$ | $\square$ | $e_0$ |
| $(e_0, \square) : \square$ | $\square$ | |
| $\square$ | $(e_0, \square) : \square$ | $e_1$ |
| $(e_1, (e_0, \square) : \square) : \square$ | $(e_0, \square) : \square$ | |
| $\square$ | $(e_1, (e_0, \square) : \square) : (e_0, \square) : \square$ | $e_2$ |
| $(e_2, (e_1, (e_0, \square) : \square) : (e_0, \square) : \square) : \square$ | $(e_1, (e_0, \square) : \square) : (e_0, \square) : \square$ | |
| $\square$ | $(e_2, (e_1, (e_0, \square) : \square) : (e_0, \square) : \square)$ $: (e_1, (e_0, \square) : \square) : (e_0, \square) : \square$ | $e_3$ |

(b) Trace of the token data (selected), until the token reaches $e_3$ for the first time

Figure 2.19: Example execution of the jumping machine: exponential growth of the environment stack

the token follows the shortest (directed) path to $e_3$, visiting $D$-nodes and !-nodes alternatively. First, the token passes a $D$-node. It pushes the input link $e$ of the $D$-node and a copy of the whole environment stack $E$ (indicated by magenta in Fig. 2.19(b)) onto the box stack. Immediately afterwards, the token passes a !-node. It moves the new top element $(e, E)$ of the box stack to the top of the environment stack $E$. As a result, the environment stack $E$ has become $(e, E) : E$, doubling in size and additionally gaining the extra element $e$.

By the time the token reaches the link $e_3$, the token repeats this doubling procedure three times in a row, and hence the environment stack exponentially grows. Note that the token does not consume any element of the stack meanwhile. The token does not visit any ?-nodes nor perform any jump until it reaches $e_3$. It is the nested $\eta$-expansions of the term that creates this behaviour here.

# Chapter 3

# Focussed graph rewriting for the core calculus Spartan

## 3.1 Outline

This chapter presents a semantical framework with which observational equivalence can be proved directly by exploiting the idea of locality, the idea that naturally arises in token-guided graph rewriting. We revise the rewrites-first DGoIM presented in Chap. 2 for the lambda-calculus, into the *Universal Abstract Machine* (UAM) for the core untyped language Spartan. All language features but three—variable binding, name binding and thunking—have to be provided as extrinsic operations to Spartan, with their behaviour provided as extrinsic graph-rewrite rules to the UAM. This means that it is not necessary to classify features into intrinsic and extrinsic, or into effects and pure computation; in fact, even the function-abstraction and application are modelled as extrinsic operations in our framework.

This chapter is organised as follows. In Sec. 3.2 we present a simple calculus (Spartan) which intermediates syntactically between languages with effects and the focussed hypernet representation. Hypernet rewriting is overviewed in Sec. 3.3, with enough technical detail to understand the examples of observational equivalence listed in Sec. 3.4. The technical details of focussed hypernet rewriting, including the

definition of the UAM, are given in Sec. 3.5. Execution of the UAM, given a variety of pre-loaded extrinsic operations, can be seen in an on-line visualiser[1].

## 3.2 The Spartan calculus

The syntactic elements of SPARTAN are a set of *variables* $\mathbb{V}$, (ranged over by $x, y$), a set of *atoms* $\mathbb{A}$, (ranged over by $a$), and a set of *operations* $\mathbb{O}$ (ranged over by $\phi$). The set of operations is partitioned into $\mathbb{O} = \mathbb{O}_{\checkmark} \uplus \mathbb{O}_{\lightning}$, namely: *passive* operations $\mathbb{O}_{\checkmark}$ (ranged over by $\phi_{\checkmark}$) and *active* operations $\mathbb{O}_{\lightning}$ (ranged over by $\phi_{\lightning}$). We will usually denote a sequence of variables $x_0, \ldots, x_k$ by $\vec{x}$, a sequence of atoms $a_0, \ldots, a_k$ by $\vec{a}$, etc. These sequences may be empty, case in which we write $-$. We denote the length of a sequence $\vec{x}$ by $|\vec{x}|$. If convenient we may treat the sequences as sets; for example, we write $y \in \vec{x}$ if $y$ appears in the sequence $\vec{x}$, and we denote the set of common elements between two sequences $\vec{x}$ and $\vec{y}$ by $\vec{x} \cap \vec{y}$.

**Definition 3.2.1** (SPARTAN terms)**.** The terms of SPARTAN, typically ranged over by $s, t, u$, are generated by the grammar:

$$t ::= x \mid a \mid \texttt{new}\ a \multimap t\ \texttt{in}\ t \mid \texttt{bind}\ x \to t\ \texttt{in}\ t \mid \vec{y}.t \mid \phi(\vec{t}; \vec{t}).$$

The term formers are variables, names, name binding, variable binding, thunking and operations. Note that the sequences above may be empty. In particular, thunking may be applied without binding any variable ($-.t$), in which case we may simply write $t$. In the formation of an operation term $\phi(\vec{s}; \vec{t})$ arguments $\vec{s}$ are used eagerly and the arguments $\vec{t}$ lazily (we also say they are *deferred*). The eager arguments are evaluated in the given order, whereas the evaluation of lazy arguments is deferred. The distinction between name-binding and variable-binding will be seen to play a crucial role in the management of sharing vs. copying during execution.

[1]Link to the on-line UAM visualiser: `https://tnttodda.github.io/Spartan-Visualiser/`

The calculus provides only the most basic infrastructure. All interesting computational content must be concretely provided as extrinsic operations. The following is a non-exhaustive list of such operations which may be added to the SPARTAN framework. These operations can be used to write SPARTAN terms that recursively compute the factorial of five, for example, as shown in Fig. 3.1.

Some passive operations (constructors) that can be added to SPARTAN are numerical constants ($n \stackrel{def}{=} n(-;-)$), boolean constants ($\mathtt{tt} \stackrel{def}{=} \mathtt{tt}(-;-)$ and $\mathtt{ff} \stackrel{def}{=} \mathtt{ff}(-;-)$), pairing ($\langle t, u\rangle \stackrel{def}{=} \mathtt{P}(t, u; -)$), empty tuple ($\langle\rangle \stackrel{def}{=} \langle\rangle(-;-)$), injections ($\mathtt{inj}_i(t) \stackrel{def}{=} \mathtt{inj}_i(t;-), i \in \{1,2\}$), etc.

Some examples of active SPARTAN operations are successor ($\mathtt{succ}(t) \stackrel{def}{=} \mathtt{succ}(t;-)$), addition ($t + u \stackrel{def}{=} +(t, u; -)$), conjunction ($t \,\&\, u \stackrel{def}{=} \&(t, u; -)$), conditionals ($\mathtt{if}\ t\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2 \stackrel{def}{=} \mathtt{if}(t; u_1, u_2)$), recursion ($\mu x.t \stackrel{def}{=} \mu(-; x.t)$), sequential composition ($t; u \stackrel{def}{=} ;(t; u)$ where the operation '$;$' is distinguished from the punctuation ';') etc.

Datatype destructors can also be added as active operations, for example projections ($\pi_i(t) \stackrel{def}{=} \pi_i(t;-)$, $i \in \{1,2\}$), pattern matching ($\mathtt{match}\ t\ \mathtt{with}\ x_1 \mapsto u_1, x_2 \mapsto u_2 \stackrel{def}{=} \mathtt{match}(t; x_1.u_1, x_2.u_2)$), etc.

Operations with multiple arguments can come in different flavours depending on order of evaluations and eagerness. For example conjunction can be left-to-right $\&(t, u; -)$, right-to-left $\&(u, t; -)$, or short-cut $\&(t; u)$ that evaluates $u$ only when the evaluation result of $t$ is true ($\mathtt{tt}$). Pairs can be also evaluated left-to-right (as above), but also, right-to-left, or lazily.

Most strikingly, abstraction and application themselves can be presented as operations. Abstraction is simply a thunking of the function body: $\lambda x.t \stackrel{def}{=} \lambda(-; x.t)$, whereas application can be defined by-name or by-value, left-to-right or right-to-left: $t\,u \stackrel{def}{=} @(t; u)$(call-by-name), $t\,u \stackrel{def}{=} \overrightarrow{@}(t, u; -)$(left-to-right call-by-value), or $t\,u \stackrel{def}{=} \overleftarrow{@}(u, t; -)$(right-to-left call-by-value).

*Algebraic operations* are operations of certain arity and equational properties,

Simple recursion: with and without syntactic sugar

$$\texttt{bind}\ \mathit{fact} \to \mu f.\, \lambda x.\, \texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ x \times (f\ (x-1))\ \texttt{in}\ \mathit{fact}\ 5$$

$$\begin{aligned}\texttt{bind}\ \mathit{fact} \to\ & \boldsymbol{\mu}(\text{-}; f.\, \boldsymbol{\lambda}(\text{-}; x.\, \texttt{if}(=(x, \mathbf{1}(\text{-};\text{-}); \text{-}); \\ & \qquad \text{-}.\mathbf{1}(\text{-};\text{-}), \\ & \qquad \text{-}.\times(x, \overrightarrow{@}(f, -(x, \mathbf{1}(\text{-};\text{-}); \text{-}); \text{-}); \text{-})))) \\ \texttt{in}\ & \overrightarrow{@}(\mathit{fact}, \mathbf{5}(\text{-};\text{-}); \text{-})\end{aligned}$$

Tail recursion using pairs: with and without syntactic sugar

$$\begin{aligned}\texttt{bind}\ \mathit{factT} \to\ & \mu f.\, \lambda n.\, \texttt{if}\ \pi_2(n) = 1\ \texttt{then}\ \pi_1(n) \\ & \qquad \texttt{else}\ f\ \langle \pi_1(n) \times \pi_2(n),\ \pi_2(n) - 1 \rangle \\ \texttt{in}\ & \mathit{factT}\ \langle 1, 5 \rangle\end{aligned}$$

$$\begin{aligned}\texttt{bind}\ \mathit{factT} \to\ & \boldsymbol{\mu}(\text{-}; f.\, \boldsymbol{\lambda}(\text{-}; n.\, \texttt{if}(=(\boldsymbol{\pi_2}(n; \text{-}), \mathbf{1}(\text{-};\text{-}); \text{-}); \\ & \qquad \text{-}.\boldsymbol{\pi_1}(n; \text{-}), \\ & \qquad \text{-}.\overrightarrow{@}(f, \texttt{P}(\times(\boldsymbol{\pi_1}(n; \text{-}), \boldsymbol{\pi_2}(n; \text{-}); \text{-}), \\ & \qquad\qquad -(\boldsymbol{\pi_2}(n; \text{-}), \mathbf{1}(\text{-};\text{-}); \text{-}); \text{-}); \text{-})))) \\ \texttt{in}\ & \overrightarrow{@}(\mathit{factT}, \texttt{P}(\mathbf{1}(\text{-};\text{-}), \mathbf{5}(\text{-};\text{-}); \text{-}); \text{-})\end{aligned}$$

Recursion using state: with and without syntactic sugar

$$\begin{aligned}\texttt{bind}\ \mathit{factS} \to\ & \texttt{new}\ a \multimap 1\ \texttt{in} \\ & \mu f.\, \lambda x.\, \texttt{if}\ x = 1\ \texttt{then}\ !a\ \texttt{else}\ ((a := !a \times x); f\ (x-1)) \\ \texttt{in}\ & \mathit{factS}\ 5\end{aligned}$$

$$\begin{aligned}\texttt{bind}\ \mathit{factS} \to\ & \texttt{new}\ a \multimap \mathbf{1}(\text{-};\text{-})\ \texttt{in} \\ & \boldsymbol{\mu}(\text{-}; f.\, \boldsymbol{\lambda}(\text{-}; x.\, \texttt{if}(=(x, \mathbf{1}(\text{-};\text{-}); \text{-}); \\ & \qquad \text{-}.\texttt{deref}(a; \text{-}), \\ & \qquad \text{-}.;(\texttt{assign}(a, \times(\texttt{deref}(a; \text{-}), x, ; \text{-}); \text{-}); \\ & \qquad\quad \text{-}.\overrightarrow{@}(f, -(x, \mathbf{1}(\text{-};\text{-}); \text{-}); \text{-}))))) \\ \texttt{in}\ & \overrightarrow{@}(\mathit{factS}, \mathbf{5}(\text{-};\text{-}); \text{-})\end{aligned}$$

Figure 3.1: SPARTAN terms with extrinsic operations (highlighted in magenta)

designed to characterise a class of computational effects [Plotkin and Power, 2003]. All examples of algebraic operations from *loc. cit.* can be presented as (active) SPARTAN operations. Non-deterministic choice $\sqcup(-; u_1, u_2)$ selects one of its deferred argument non-deterministically; equipping it with a probability $p \in [0, 1]$ yields a probabilistic choice $\sqcup_p(-; u_1, u_2)$. Interactive input $\texttt{read}(-; x.u)$ assigns an input value to bound variable $x$, whereas interactive output $\texttt{write}(t; u)$ produces the result of evaluating (eager) argument $t$ as an output value. Both operations continue with the deferred argument $u$. State lookup and update take form $\texttt{lookup}(t_0; x.u)$ and, respectively, $\texttt{update}(t_0, t; u)$. The difference from interactive I/O is that the first eager argument $t_0$ is evaluated to yield an atom, which serves as a location in a store. In their generic form, state operations $!t_0 \stackrel{def}{=} \texttt{deref}(t_0; -)$ and $t_0 := t \stackrel{def}{=} \texttt{assign}(t_0, t; -)$ return a stored value and the empty pair, respectively, instead of continuing with a deferred argument. Finally, creation of a local state, in the generic form, is modelled by $\texttt{ref}(t; -)$ that stores evaluation result of $t$ to a fresh location (atom) and returns it.

*Effect handlers* are studied as a generalisation of exception handlers to algebraic effects in Plotkin and Pretnar [2013]. In our setting, a handler that targets operations $\phi_1, \ldots, \phi_m$ can be constructed by a passive operation $\texttt{handler}_{\phi_1,\ldots,\phi_m}$ tagged with the targets. It is natural to assume that targeted operations are never passive (e.g. lambda-abstraction, pairing operation, injection, and handlers themselves). These handler constructors are a (rare) example in which a deferred argument may bind more than one variables. A handler can be then used with the handling operation $\texttt{with\_handle}(t, u; -)$ to evaluate $u$ with handler $t$.

*Control operators* are similarly dealt with. The undelimited control operator `call/cc` can be defined as:

$$\texttt{callcc}(t) \stackrel{def}{=} \texttt{callcc}(-; t), \quad \texttt{abort}(t) \stackrel{def}{=} \texttt{abort}(-; t)$$

The delimited control operator `shift`/`reset` of Danvy and Filinski [1990] can be defined as:

$$\texttt{shift}(t) \stackrel{def}{=} \texttt{shift}(-;t), \quad \texttt{reset}(t) \stackrel{def}{=} \texttt{reset}(t;-).$$

### 3.2.1 The Spartan type system

The type system is primarily used to distinguish thunks from eager terms and to ensure terms are well formed. However, this type system does not enforce any notion of run-time safety.

Terms are defined by $\tau ::= \star \mid T^m(\star)$ where $m \in \mathbb{N}$. Eager terms have type $\star$ and thunks have type $T^m(\star)$ for some $m \in \mathbb{N}$, representing the number of bound variables that accompany the thunk. Note that $T^0(\star)$ is a valid type and that $T^0(\star) \neq \star$. A sequence $\tau_1, \ldots, \tau_n$ of types is denoted by an expression $\otimes_{i=1}^n \tau_i$. This is empty if $n = 0$, and equal to a single type if $n = 1$, i.e. $\otimes_{i=1}^1 \tau = \tau$ . The empty sequence is written as $\epsilon$. We use syntactic sugar $\tau^{\otimes n} \equiv \otimes_{i=1}^n \tau$, and $\tau^{\otimes n} \otimes \tau^{\otimes n'} = \tau^{\otimes (n+n')}$.

Each operation $\phi \in \mathbb{O}$ has an *arity*, given in the form of $\phi \vdash (m_{\mathsf{e}}; \vec{n}) \in \mathbb{N} \times \mathbb{N}^{m_{\mathsf{d}}}$. It indicates that the operation $\phi$ takes $m_{\mathsf{e}}$ eager arguments and $m_{\mathsf{d}}$ thunks. The sequence $\vec{n}$ specifies the number of bound variables each thunk expects.

For a type $\tau$ and a term $t$, judgements are written as $\vec{x} \mid \vec{a} \vdash t : \tau$, where all variables in $\vec{x}$ and all atoms in $\vec{a}$ are distinct. The rules are given in Fig. 3.2, where $|\vec{x}| = k, |\vec{y}| = n, |\vec{a}| = h$. If $- \mid - \vdash t : \star$ is derivable, term $t$ is called a *program*.

### 3.2.2 Semantics of bindings: copying vs. sharing

We opt for simple semantics of bindings, that is, bound computation ($u$ in `bind` $x \to u$ `in` $t$ or `new` $a \multimap u$ `in` $t$) is *not* evaluated eagerly. In variable binding `bind` $x \to u$ `in` $t$, moreover, the bound computation $u$ is evaluated as many times as the variable $x$ is required in $t$. The eager version of variable binding, where $u$ is always evaluated exactly once, can be actually included in SPARTAN as a single extrinsic operation $\texttt{let}(u; x.t)$. On the other hand, the lazy version, where $u$ is evaluated at most

$$\frac{}{\vec{x} \mid \vec{a} \vdash x_i : \star}\ 1 \leq i \leq k \qquad \frac{}{\vec{x} \mid \vec{a} \vdash a_j : \star}\ 1 \leq j \leq h$$

$$\frac{\vec{x} \mid \vec{a} \vdash u : \star \quad \vec{x}, x \mid \vec{a} \vdash t : \star}{\vec{x} \mid \vec{a} \vdash \texttt{bind}\ x \to u\ \texttt{in}\ t : \star}\ x \notin \vec{x} \qquad \frac{\vec{x} \mid \vec{a} \vdash u : \star \quad \vec{x} \mid a, \vec{a} \vdash t : \star}{\vec{x} \mid \vec{a} \vdash \texttt{new}\ a \multimap u\ \texttt{in}\ t : \star}\ a \notin \vec{a}$$

$$\frac{\vec{y}, \vec{x} \mid \vec{a} \vdash t : \star}{\vec{x} \mid \vec{a} \vdash \vec{y}.t : T^n(\star)}\ n = |\vec{y}|, \vec{x} \cap \vec{y} = \emptyset$$

$$\frac{\phi \vdash (m; \vec{n}) \in \mathbb{N} \times \mathbb{N}^p \quad \{\vec{x} \mid \vec{a} \vdash t_i : \star\}_{i=1}^{m} \quad \{\vec{x} \mid \vec{a} \vdash s_j : T^{n_j}(\star)\}_{j=1}^{p}}{\vec{x} \mid \vec{a} \vdash \phi(\vec{t}; \vec{s}) : \star}$$

Figure 3.2: The SPARTAN type system

once, would require combination of extrinsic operations (*à la* CPS transformation for the call-by-need lambda-calculus [Okasaki et al., 1994]) or an additional intrinsic mechanism of SPARTAN such as memoisation.

Roughly speaking, the two kinds of binding in SPARTAN, namely variable binding and name binding, deal with copying and sharing of computation respectively. In variable binding $\texttt{bind}\ x \to u\ \texttt{in}\ t$, the bound variable $x$ is expected to be substituted by its associated term $u$. This means that multiple occurrences of $x$ in $t$ would require copying of $u$. On the other hand, in name binding $\texttt{new}\ a \multimap u\ \texttt{in}\ t$, the bound name $a$ is expected to act merely as the unique reference to (or literally, the unique name of) its associated term $u$. Even if the bound name $a$ occurs multiple times in $t$, its associated computation $u$ remains shared. The shared computation $u$ can only be accessed by extrinsic operations, such as $\texttt{deref}(t_0; -)$ and $\texttt{assign}(t_0, t; -)$ that we mentioned earlier, using the name $a$.

The copying behaviour of variable binding and the sharing behaviour of name binding are combined in SPARTAN, in such a way that shared computation is never copied unless it is deferred. In other words, the copying power of variable binding is restricted, so that it does not violate sharing that is already caused by name binding. We illustrate this below, using the aforementioned extrinsic operations $\lambda$ and $\overrightarrow{@}$ for the lambda-calculus. What determines the restriction of duplication is where name

binding appears, namely whether name binding appears inside a thunk (i.e. deferred computation).

Firstly, in a term $\texttt{bind}\ x \to (\texttt{new}\ a \multimap u\ \texttt{in}\ \lambda(-; y.t))\ \texttt{in}\ \overrightarrow{@}(x, x; -)$, the name binding $\texttt{new}\ a \multimap u\ \texttt{in}\ \lambda(-; y.t)$ appears outside any thunk, and hence is not deferred. This means that the computation $u$ is already shared, and is never copied by variable binding. Reduction of the term would be as below:

$$\begin{aligned} &\texttt{bind}\ x \to (\texttt{new}\ a \multimap u\ \texttt{in}\ \lambda(-; y.t))\ \texttt{in}\ \overrightarrow{@}(x, x; -) \\ &\rightsquigarrow\ \texttt{new}\ a \multimap u\ \texttt{in}\ \texttt{bind}\ x \to \lambda(-; y.t)\ \texttt{in}\ \overrightarrow{@}(\lambda(-; y.t), \lambda(-; y.t); -) \end{aligned} \tag{3.1}$$

where the variable binding for $x$ can only copy the term $\lambda(-; y.t)$. The shared computation $u$ is kept shared via the unique name $a$ across all the copies of the term $\lambda(-; y.t)$. Note that the type system ensures that the bound variable $x$ does not appear in the shared computation $u$, which allows the name binding and the variable binding to be commutative.

On the other hand, in a term $\texttt{bind}\ x \to \lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t))\ \texttt{in}\ \overrightarrow{@}(x, x; -)$, the name binding $\texttt{new}\ a \multimap u\ \texttt{in}\ t$ appears inside a thunk. Sharing of the computation $u$ is considered to be delayed accordingly, and reduction of the term would be as follows:

$$\begin{aligned} &\texttt{bind}\ x \to \lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t))\ \texttt{in}\ \overrightarrow{@}(x, x; -) \\ &\rightsquigarrow\ \texttt{bind}\ x \to \lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t))\ \texttt{in} \\ &\quad \overrightarrow{@}(\lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t)), \lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t)); -). \end{aligned} \tag{3.2}$$

In contrast to reduction (3.1), the variable binding for $x$ is allowed to copy the whole term $\lambda(-; y.(\texttt{new}\ a \multimap u\ \texttt{in}\ t))$ including name binding.

Although it would be possible to define reduction semantics of SPARTAN, we leave it for future work and opt for *focussed graph-rewriting* semantics. The fo-

cussed graph-rewriting semantics can model the subtle interplay between copying and sharing, namely the rather complicated behaviour of variable binding, in a simpler way than the reduction semantics. This is due to the name-free representation of terms as graphs where bindings are maintained simply by connections.

Representing terms as graphs is crucial also to develop a proof methodology of observational equivalence, in which one can directly analyse how sub-terms evolve during evaluation. The analysis can be done simply by looking at sub-graphs, which may not represent a term. For example, in reduction (3.1) above, the name binding $\texttt{new}\ a \multimap u\ \texttt{in}\ \lambda(-; y.t)$ syntactically evolves into two copies of the sub-term $\lambda(-; y.t)$ and a string "$\texttt{new}\ a \multimap u\ \texttt{in}$" that is not itself a sub-term. In the corresponding focussed graph-rewriting, however, the graph representation of $\texttt{new}\ a \multimap u\ \texttt{in}\ \lambda(-; y.t)$ would evolve into three sub-graphs that represent the sub-terms $\lambda(-; y.t)$, $\lambda(-; y.t)$, and the string "$\texttt{new}\ a \multimap u\ \texttt{in}$". This suggests that the direct analysis is difficult by means of *sub-terms*, but instead possible by means of *sub-graphs*.

## 3.3 Overview of focussed graph rewriting

### 3.3.1 Monoidal hypergraphs and hypernets

Given a set $X$ we write by $X^*$ the set of elements of the free monoid over $X$. Given a function $f : X \to Y$ we write $f^* : X^* \to Y^*$ for the point-wise application (map) of $f$ to the elements of $X^*$.

**Definition 3.3.1.** A *monoidal hypergraph* is a pair $(V, E)$ of finite sets, *vertices* and *(hyper)edges*, along with a pair of functions $S : E \to V^*$, $T : E \to V^*$ defining the *source list* and *target list*, respectively, of an edge.

**Definition 3.3.2.** A *labelled monoidal hypergraph* consists of a monoidal hypergraph, a set of vertex labels $L$, a set of edge labels $M$, and labelling functions $f_{\mathsf{V}} : V \to L, f_{\mathsf{E}} : E \to M$ such that:

- For any edge $e \in E$, its source list $S(e)$ consists of distinct vertices, and its target list $T(e)$ also consists of distinct vertices.
- For any vertex $v \in V$ there exists at most one edge $e \in E$ such that $v \in S(e)$ and at most one edge $e' \in E$ such that $v \in T(e')$.
- For any edges $e_1, e_2 \in E$ if $f_{\mathsf{E}}(e_1) = f_{\mathsf{E}}(e_2)$ then $f^*_{\mathsf{V}}\big(S(e_1)\big) = f^*_{\mathsf{V}}\big(S(e_2)\big)$, and $f^*_{\mathsf{V}}\big(T(e_1)\big) = f^*_{\mathsf{V}}\big(T(e_2)\big)$.

In words, the label of an edge is always consistent with the number and labelling of its endpoints. This makes it possible to use the vertex labels as *types* for edge labels. Given $m \in M$ we can write $m : x \Rightarrow x'$ for $x, x' \in L^*$ such that $x = f^*_{\mathsf{V}}\big(S(e)\big)$ and $x' = f^*_{\mathsf{V}}\big(T(e)\big)$ for any $e \in E$ such that $f_{\mathsf{E}}(e) = m$.

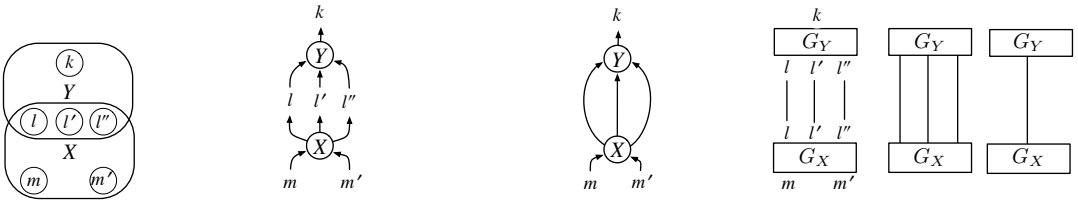
If a vertex belongs to the target (resp. source) list of no edge we call it an *input* (resp. *output*).

**Definition 3.3.3.** A labelled monoidal hypergraph is *interfaced* if inputs and outputs are respectively ordered, and no vertex is both an input and an output.

We can type an interfaced hypergraph $G : f^*_{\mathsf{V}}(I) \Rightarrow f^*_{\mathsf{V}}(O)$ where $I, O$ are the lists of inputs and outputs, respectively. Note that a single hypergraph can be interfaced, and hence typed, in several ways. Interfaces (i.e. pairs of a list of inputs and a list of outputs) of the same hypergraph are given by permutations of inputs and permutations of outputs.

**Definition 3.3.4** (Interface permutation)**.** Let $G$ be a hypergraph with an input list $i_1, \ldots, i_n$ and an output list $o_1, \ldots, o_m$. Given two bijections $\rho$ and $\rho'$ on sets $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, respectively, we write $\Pi^{\rho'}_{\rho}(G)$ to denote the hypergraph that is defined by the same data as $G$ except for the input list $i_{\rho(1)}, \ldots, i_{\rho(n)}$ and the output list $o_{\rho'(1)}, \ldots, o_{\rho'(m)}$.

In the sequel, when we say hypergraphs we always mean interfaced labelled monoidal hypergraphs.




(a) Hypergraph (b) Graph representation (c) Eliding some nodes (d) Identity lines

Figure 3.3: Graphical conventions

**Definition 3.3.5** (Hypernets)**.** Given a set of vertex labels $L$ and edge labels $M$ we write $\mathcal{H}(L, M)$ for the set of hypergraphs with these labels; we also call these *level-0 hypernets* $\mathcal{H}_0(L, M)$. We call *level-*$(k{+}1)$ *hypernets* the set of hypergraphs

$$\mathcal{H}_{k+1}(L, M) = \mathcal{H}\Big(L, M \cup \bigcup_{i \leq k} \mathcal{H}_i(L, M)\Big).$$

We call (labelled monoidal) *hypernets* the set $\mathcal{H}_\omega(L, M) = \bigcup_{i \in \mathbb{N}} \mathcal{H}_i(L, M)$.

Informally, hypernets are nested hypergraphs, up to some finite depth, using the same sets of labels. An edge labelled with a hypergraph is called *box* edge, and a hypergraph labelling a box edge is called *content*. Edges of a hypernet $G$ are said to be *shallow*. Edges of nesting hypernets of $G$, i.e. edges of hypernets that recursively appear as edge labels, are said to be *deep* edges of $G$. Shallow edges and deep edges of a hypernet are altogether referred to as edges *at any depth*.

### 3.3.2 Graphical conventions

A monoidal hypergraph $G$ with vertices $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ and edges $E = \{e_0, e_1\}$ such that

$$S(e_0) = \{v_0, v_1\}$$
$$T(e_0) = S(e_1) = \{v_2, v_3, v_4\}$$
$$T(e_1) = \{v_5\}$$

$$f_V = \{v_0 \mapsto m, v_1 \mapsto m', v_2 \mapsto l, v_3 \mapsto l', v_4 \mapsto l'', v_k \mapsto k\}$$
$$f_E = \{e_0 \mapsto X, e_1 \mapsto Y\}$$

is normally represented as Fig. 3.3a. However, this style of representing hypergraphs is awkward for understanding their structure. We will often graphically represent hypergraphs as graphs, like Fig. 3.3b, with both vertices and edges drawn as nodes marked by their labels and connecting input vertices with edges, and edges with output vertices using arrows. To distinguish them, the edge labels are circled.

Since the node labels are often determined in our typed graphs by the edges we can omit them to avoid clutter, showing only the edges and the way they link. The graph $G$ would be drawn like Fig. 3.3c.

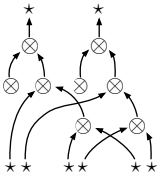
Sometimes we will draw a hypergraph so that to identify sub-graphs of interest. In this case we may draw interface nodes twice and connect them with arrowless lines which indicate *identity*, i.e. the nodes at either ends are two graphical representations of the same node. If it is obvious from context we may just draw the identity lines between the sub-graphs, or just one line if the entire input interface of a sub-graph is identified with the entire output interface of another sub-graph. For example, the graph $G$ can be drawn like Fig. 3.3d, where $G_X$ and $G_Y$ are the sub-graphs consisting just of edge $e_0$ (labelled with $X$) and $e_1$ (labelled with $Y$), respectively.

The final conventions are that a bunch of parallel arrows can be drawn as a single arrow with a dash across, and that a hypergraph-labelled edge is indicated with a dotted box, and decorated with its type.

### 3.3.3 From terms to hypernets

We represent terms in SPARTAN as hypernets with vertex labels either $\tau$ or $\diamond$, ranged over by $\ell$. The same notational conventions apply, extended with the extra symbol $\diamond$. The edge labels have unique types, and they are: the operations $\phi$ (with the edge-label-type inherited from the SPARTAN type system), an *instance* label $\mathsf{I} : \star \Rightarrow \diamond$, an *atom* label $\circ : \diamond \Rightarrow \star$, a family of *weakening* and *contraction* labels $\{\otimes^{\ell}_{\mathsf{W}} \colon \epsilon \Rightarrow \ell,\ \otimes^{\ell}_{\mathsf{C}} \colon \ell^{\otimes 2} \Rightarrow \ell \mid \ell \in \{\star, \diamond\}\}$, a family of *token* labels $?, \checkmark, \lightning : \star \Rightarrow \star$. When a hypergraph is used as an edge label it must always have type $G : \star \Rightarrow \star^{\otimes n} \otimes \diamond^{\otimes h}$; the box edge is assigned type $T^{n-k}(\star) \Rightarrow \star^{\otimes k} \otimes \diamond^{\otimes h}$ for some $k \leq n$.

The hypernet is said to be *focussed* if: there exists only one token-labelled edge, the token-labelled edge is shallow, and the hypernet satisfies some other basic well-formedness conditions discussed later.

To graphically represent multiple occurrences of a single variable or a single atom, we will define and use a family of sub-graphs $D^{\ell}_{k,m} : \ell^{\otimes km} \Rightarrow \ell^{\otimes k}$ with $\ell \in \{\diamond, \star\}$ which we call *distributors* (Def. 3.5.6). Intuitively, they are the $k$-interleaving of bunches of $m$-contraction edges, for instance, $D^{*}_{2,3} =$  and $D^{\diamond}_{3,0} =$  .

The translation function $(-)^{\dagger}$ from SPARTAN terms to hypernets is given in Fig 3.4, where $\boxtimes_{i=1}^{m} G_i$ is a hypernet formed of a family of hypernets $(G_i : \star \Rightarrow \star^{\otimes k_i} \otimes \diamond^{\otimes h_i})$ placed side by side. We can easily see that the hypernet of a SPARTAN program always has type $(- \mid - \vdash t : \star)^{\dagger} : \star \Rightarrow \epsilon$, i.e. one input and no outputs. The hypernet of a program is focussed by adding a ? token in the input position.

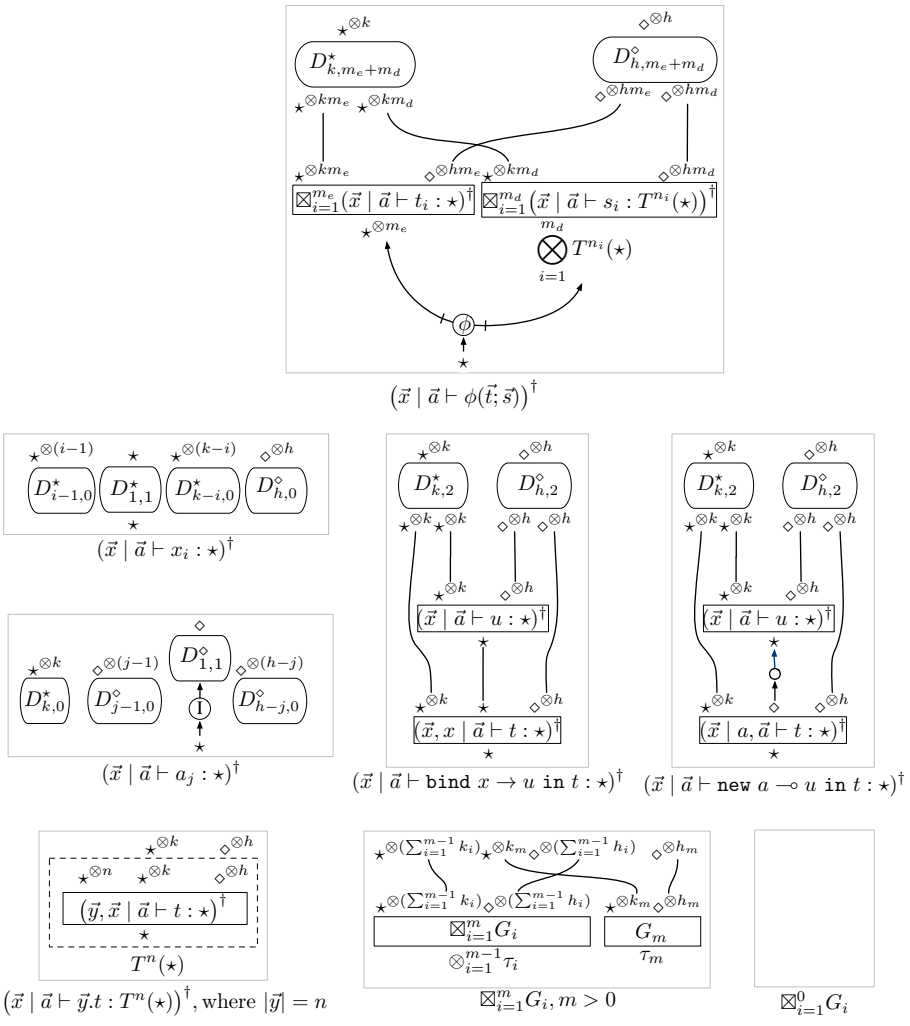



Figure 3.4: Hypernet semantics of SPARTAN

### 3.3.4 Focussed rewriting

In this section we illustrate an abstract machine for rewriting hypernets of SPARTAN terms, thus giving a notion of evaluation for the calculus. The machine, dubbed *Universal Abstract Machine* (UAM), is based on the DGoIM presented in Chap. 2, which it generalises for the modelling of effects. The full technical details are presented later, in Sec. 3.5.

In a conventional reduction semantics a subtle and often complex aspect of the reduction bureaucracy is the identification and isolation of a sub-term which is a *redex* from the *context* [Felleisen and Hieb, 1992]. Similarly, in an abstract machine much of the work consists of moving information (representing the context) on and off the stack in order to reach redexes [Wright and Felleisen, 1994]. In both cases the formal machinery identifies an implicit *focus* of action which either moves around the term or is acted upon via a substitution. In the case of focussed graph rewriting the focus is represented by the token edge. The redex search is defined by the local interaction between the token and the neighbouring edges, which can cause the token to navigate the graph. The same local interactions can determine a rewrite action.

The state of the token edge determines the possible actions. If the token is ? then the applicable rules (*interaction rules*) are search rules, which do not change the underlying graph; the ? token always travels in the direction of the edges and it is searching for a redex. In comparison to a conventional reduction semantics this is a narrowing of the term-in-context. If the token is ✓ then the applicable rules are also search (interaction) rules but the token travels against the direction of the edges. In comparison to a reduction semantics this is a widening of the term-in-context normally following the detection of a value. Finally, if the token is $\lightning$ then a rewrite is about to be performed at the very next step.

Interaction rules are in Fig. 3.5. These rules capture the intuition behind redex search discussed earlier:

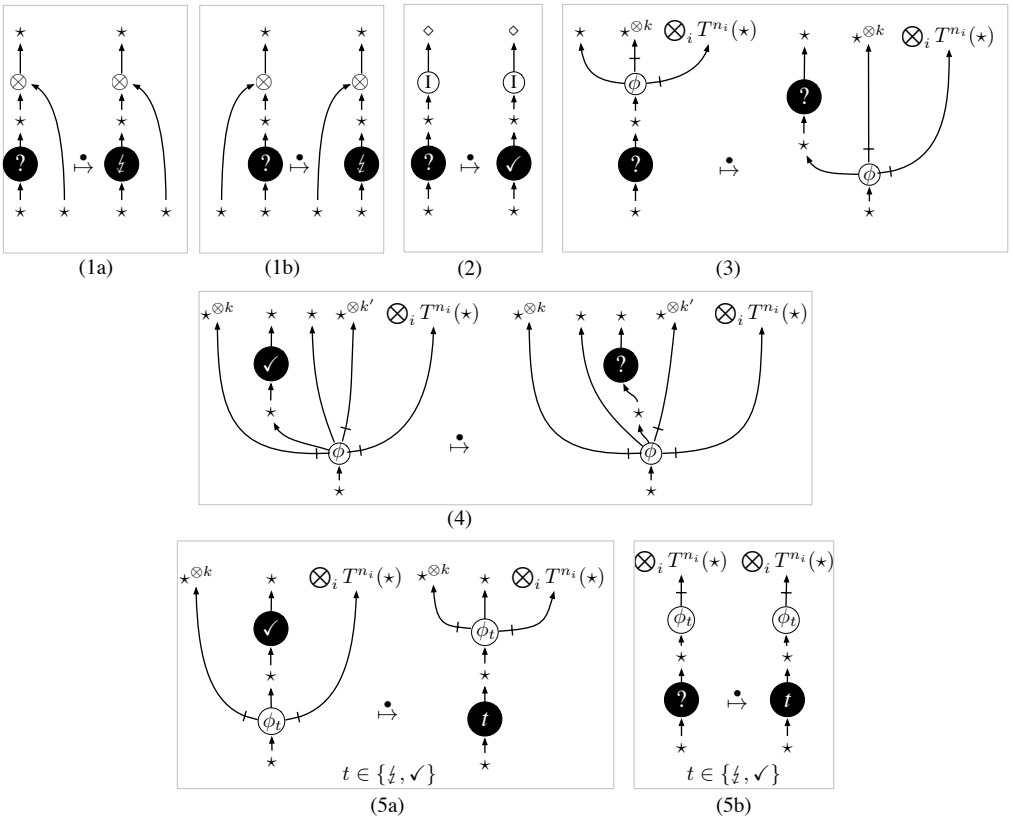



Figure 3.5: Interaction rules

**Rule 1.** When encountering a contraction ($\otimes$) the search token (?) becomes a rewrite token ($\lightning$) as a copying action will follow. This rule has two sub-rules depending on whether the focus is on the left or right branch.

**Rule 2.** When encountering an instance edge (I) the search token (?) changes to a value token ($\checkmark$) as atoms block both evaluation and copying.

**Rule 3.** When encountering an operation ($\psi$) with at least one eager argument the search token (?) will inspect the first argument.

**Rule 4.** After resolving a $(k+1)$-th eager argument of an operation ($\psi$) the value token ($\checkmark$) changes to a search token (?) which proceeds to inspect the next eager argument, if it exists.

**Rule 5a.** After all eager arguments of an operation $\psi_t$ ($t \in \{\checkmark, \lightning\}$), the token will change to a $t$-token depending on the kind of operation $\psi_t$ is, value or rewrite. (Rule 5b. is a degenerate version for operations with no eager arguments).

Once the token is $\lightning$, it triggers a rewrite. There are two kinds of rewrites: those intrinsic to the SPARTAN calculus and those extrinsic, defined by the operations. The intrinsic rewrites are only copying of sub-graphs, and they are triggered when a $\lightning$ token interacts with contraction $\otimes$. The extrinsic rewrites represent actual computation with the operations, and they are triggered when a $\lightning$ token interacts with an active operation $\phi_{\lightning}$.

Fig. 3.6 shows an example of focussed rewriting, given a hypernet with operations $\lambda(-; x.t)$ (abstraction) and $\overrightarrow{@}(t, u; -)$ (left-to-right call-by-value application). This example corresponds to an evaluation of a lambda-term $(\lambda x.x)\ (\lambda y.y)$ with the left-to-right call-by-value strategy, and it is a counterpart of the DGoIM execution presented in Fig. 2.8 (Chap. 2). In the example, steps $\overset{\bullet}{\to}$ apply the interaction rules, and steps $\to$ perform the extrinsic and intrinsic rewrites, whose actual definition will be presented in Sec. 3.4 and Sec. 3.5.4. A $\lightning$ token interacts with the application $\overrightarrow{@}$ in the first rewrite, and with contraction $\otimes$ in the second rewrite.

## 3.4 Observational equivalence

This section formulates some *desirable* observational equivalences of SPARTAN. They hold in general in well-behaved languages, but the SPARTAN framework is broad enough to allow the definition of operations that would break these and most equivalences. This is why we would refer to them for now as *desiderata*. Later in Sec. 4.5, we will show that they hold with respect to some of the operations discussed in Sec. 3.2.

Recall that our framework is parametrised by an operation set $\mathbb{O}$. Our notion of equivalence is additionally parametrised by a set $B_{\mathbb{O}}$ (*behaviour*) of extrinsic rewrites associated with the operations $\mathbb{O}$. Given two derivable judgements $\vec{x} \mid \vec{a} \vdash t_1 : \star$ and $\vec{x} \mid \vec{a} \vdash t_2 : \star$, we write $B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger_{\text{all}}} t_2 : \star)$ (resp. $B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger_{\text{all}}} t_2 : \star)$) if the hypernet of $t_1$ is a refinement (resp. equivalence) of $t_2$ in any context, with

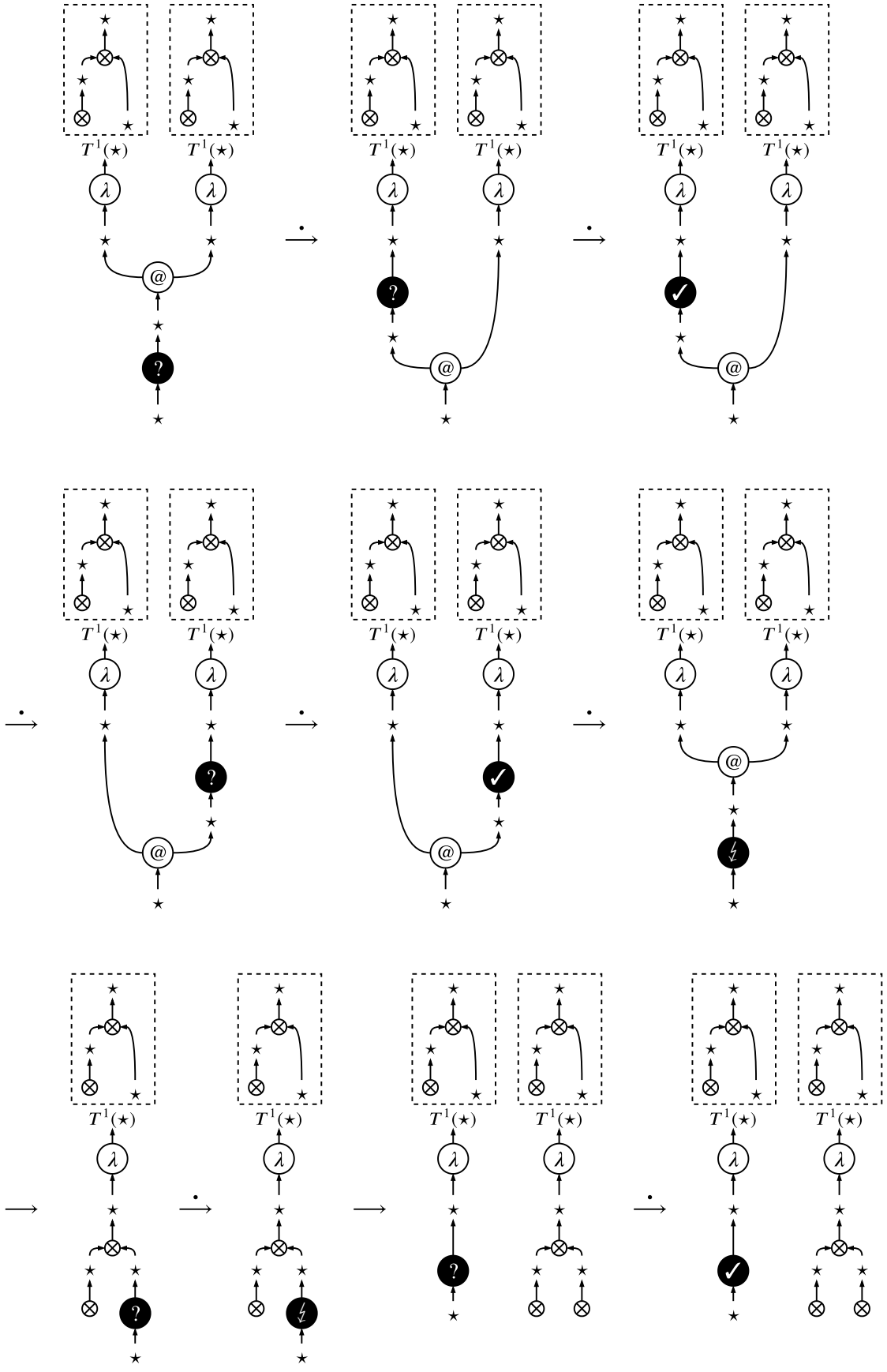



Figure 3.6: Example execution of the UAM

the operation set being $\mathbb{O}$ and extrinsic rewrites being $B_{\mathbb{O}}$. In particular, we write $B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger_{\mathrm{bf}}} t_2 : \star)$ (resp. $B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger_{\mathrm{bf}}} t_2 : \star)$), if the refinement (resp. equivalence) holds only in the contexts that do not bind a hole to a variable nor an atom. These concepts are formally defined in Sec. 4.2.1.

In the sequel, we simply write $t_1 \preceq^{\dagger_{\mathrm{all}}} t_2$ etc., omitting the type $\star$ and the sequences $\vec{x}$ and $\vec{a}$, and making the parameter $B_{\mathbb{O}}$ implicit.

### 3.4.1 Structural laws

Before considering equivalences involving specific operations, it is useful to examine *structural* equivalences, which focus on intrinsic features of SPARTAN. Let $fv(t)$ and $fa(t)$ be the sets of free variables and free atoms of a term, respectively, defined as usual.

**Desiderata 3.4.1** (Coherences)**.**

$$\texttt{bind}\ x \to t\ \texttt{in}\ \texttt{bind}\ y \to u\ \texttt{in}\ s \simeq^{\dagger_{\mathrm{all}}} \texttt{bind}\ y \to u\ \texttt{in}\ \texttt{bind}\ x \to t\ \texttt{in}\ s \quad (3.3)$$
$$(\textit{whenever}\ x \notin fv(u), y \notin fv(t))$$

$$\texttt{new}\ a \multimap t\ \texttt{in}\ \texttt{new}\ b \multimap u\ \texttt{in}\ s \simeq^{\dagger_{\mathrm{all}}} \texttt{new}\ b \multimap u\ \texttt{in}\ \texttt{new}\ a \multimap t\ \texttt{in}\ s \quad (3.4)$$
$$(\textit{whenever}\ a \notin fa(u), b \notin fa(t))$$

$$\texttt{bind}\ x \to (\texttt{bind}\ y \to u\ \texttt{in}\ t)\ \texttt{in}\ s \simeq^{\dagger_{\mathrm{all}}} \texttt{bind}\ y \to u\ \texttt{in}\ \texttt{bind}\ x \to t\ \texttt{in}\ s \quad (3.5)$$
$$(\textit{whenever}\ y \notin fv(s))$$

$$\texttt{new}\ a \multimap (\texttt{new}\ b \multimap u\ \texttt{in}\ t)\ \texttt{in}\ s \simeq^{\dagger_{\mathrm{all}}} \texttt{new}\ b \multimap u\ \texttt{in}\ \texttt{new}\ a \multimap t\ \texttt{in}\ s \quad (3.6)$$
$$(\textit{whenever}\ b \notin fa(s))$$

The second batch of laws concern *non-generative* terms defined below. Such terms are *referentially transparent*, in the sense that they can be safely replicated or substituted in other terms:

**Definition 3.4.2** ((Non-)generative terms)**.** A term is said to be *generative* if it

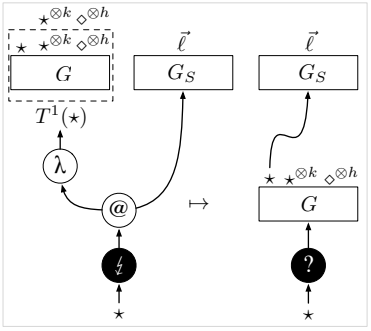



Figure 3.7: Beta rewrite rule ($G, G_S$ are hypernets)

contains a name-binder ('`new`') outside any thunks. A term $u$ is said to be *non-generative*, and written as $\overline{u}$, if all name-binders appear inside thunks.

**Desiderata 3.4.3** (Referential transparency)**.** *If $\overline{u}$ is non-generative then*

$$\texttt{bind}\ x \to \overline{u}\ \texttt{in}\ \texttt{bind}\ y \to x\ \texttt{in}\ \texttt{bind}\ z \to x\ \texttt{in}\ t$$
$$\simeq^{\dagger_{\text{all}}}\ \texttt{bind}\ y \to \overline{u}\ \texttt{in}\ \texttt{bind}\ z \to \overline{u}\ \texttt{in}\ t \quad (\textit{whenever } x \notin \mathit{fv}(t)) \tag{3.7}$$
$$\texttt{bind}\ x \to \overline{u}\ \texttt{in}\ t\ \simeq^{\dagger_{\text{all}}}\ t[\overline{u}/x] \tag{3.8}$$

To be clear, the referential transparency laws hold in contexts with most "reasonable" operations, including all mentioned in Sec. 3.4.2. However, even though referential transparency laws are very robust, one can think of operations which may violate them such as `Gc.stat` mentioned earlier.

### 3.4.2 Operations

We proceed to equivalences that involve some of the operations discussed in Sec. 3.2. First we consider operations for the lambda-calculus, namely:

$$\lambda x.t \stackrel{def}{=} \lambda(-; x.t) \qquad \text{(abstraction)}$$
$$t \stackrel{\rightarrow}{@} u \stackrel{def}{=} \stackrel{\rightarrow}{@}(t, u; -). \qquad \text{(left-to-right call-by-value application)}$$

While the abstraction $\lambda$ is a passive operation, the application $\overrightarrow{@}$ is an active operation. Fig. 3.7 shows the *beta* rewrite rule associated with $\overrightarrow{@}$. The usual observational equivalences (laws) induced by the beta rewrite are as follows.

**Desiderata 3.4.4** (Call-by-value beta laws)**.** *If $v$ is a value then*

$$(\lambda x.t)\,\overrightarrow{@}\,v \;\simeq^{\dagger_{\mathrm{bf}}}\; \texttt{bind}\; x \to v\; \texttt{in}\; t \qquad \text{(Micro beta)}$$

*Moreover, if $v$ is a referentially-transparent value then*

$$(\lambda x.t)\,\overrightarrow{@}\,v \;\simeq^{\dagger_{\mathrm{bf}}}\; t[v/x] \qquad \text{(Beta)}$$

The beta law is also robust relative to the operations described in the sequel, but reasonable operations that violate it do exist. For example, we mentioned the `Gc.stat()` function of OCaml which returns the size of the term violates this law.

Next we add some passive operations for constants (the empty pair, boolean values and integers) and active operations for arithmetic:

$$\begin{array}{rcll}
() & \stackrel{def}{=} & ()(-;-), & n \stackrel{def}{=} n(-;-), \qquad \text{(empty pair, integers)} \\
\texttt{true} & \stackrel{def}{=} & \texttt{tt}(-;-), & \texttt{false} \stackrel{def}{=} \texttt{ff}(-;-), \qquad \text{(boolean values)} \\
t+u & \stackrel{def}{=} & +(t,u;-), & t-u \stackrel{def}{=} -(t,u;-), \qquad \text{(addition, subtraction)} \\
-t & \stackrel{def}{=} & -_1(t;-). & \qquad \text{(negation)}
\end{array}$$

Fig. 3.8 shows a selected rewrite rule for arithmetic.

We further add to the language *state* (assignment and dereferencing), namely

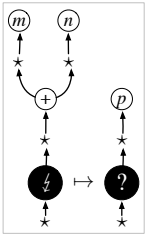



Figure 3.8: Arithmetic rewrite rule (selected), where $m, n, p \in \mathbb{N}$ and $p = m + n$

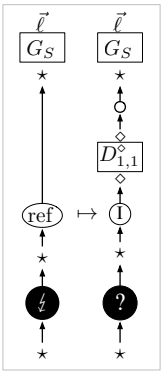



Figure 3.9: Reference creation ($G_S$ is a hypernet)

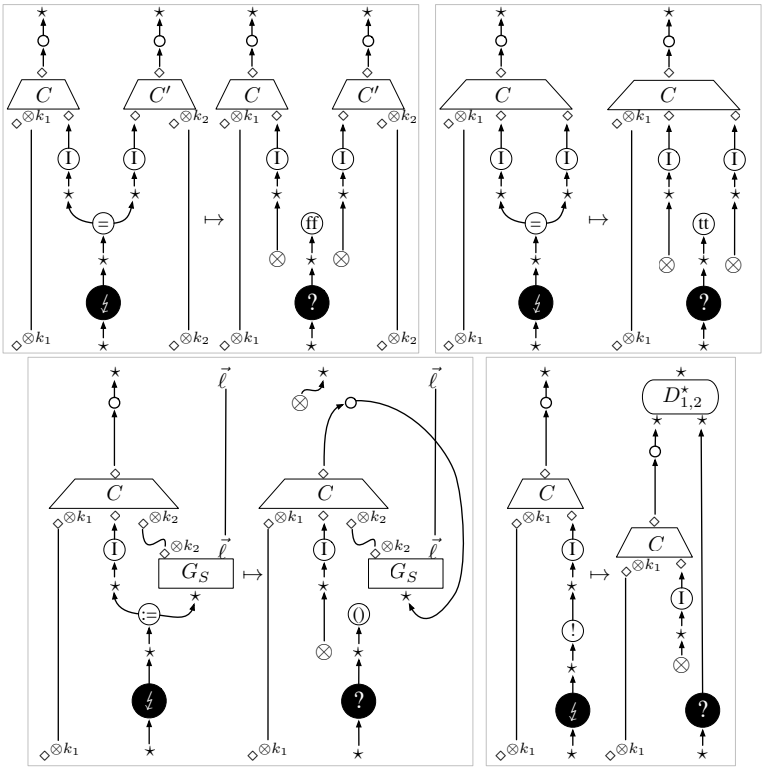



Figure 3.10: Equality, assignment, and dereferencing rewrite rules ($C$ and $C'$ are contraction trees, and $G_S$ is a hypernet)

the following four active operations:

$$\begin{aligned}
\texttt{ref}\ t &\stackrel{def}{=} \texttt{ref}(t;-) && \text{(reference creation)}\\
t = u &\stackrel{def}{=} =(t,u;-) && \text{(equality testing on references)}\\
t := u &\stackrel{def}{=} :=(t,u;-) && \text{(assignment)}\\
!t &\stackrel{def}{=} !(t;-). && \text{(dereferencing)}
\end{aligned}$$

The rewrite rule for reference creation is given in Fig. 3.9, while equality, assignment and dereferencing are given in Fig. 3.10. Since SPARTAN is (essentially) untyped, by default we consider state which can store any terms (or rather values), including lambda-abstractions.

For convenience we use the following syntactic sugar:

$$\begin{aligned}
t\ u &\stackrel{def}{=} t \mathbin{\vec{@}} u\\
\nu x.t &\stackrel{def}{=} (\lambda x.t)\ (\texttt{ref}\ ())\\
\texttt{let}\ x = u\ \texttt{in}\ t &\stackrel{def}{=} (\lambda x.t)\ u\\
\lambda_.t &\stackrel{def}{=} \lambda z.t && \text{(where } z \text{ is not a free variable in } t)\\
u; t &\stackrel{def}{=} (\lambda_.t)\ u
\end{aligned}$$

Because we are focusing on the call-by-value function application, the let-binding as a sugar has different behaviour from the intrinsic variable-binding, which has the call-by-name nature as explained in Sec. 3.2.2.

Also note that in the absence of type restrictions it is straightforward to tie recursive knots in the store. For example $\texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in}\ x := x$ stores a self-reference into $x$, so that any dereferencing $!x, !!x, !!!x$, etc., will always produce the same result.

Some desirable equivalences which hold in contexts containing stateful operations are:

**Desiderata 3.4.5** (Stateful laws)**.**

$$\nu z.\lambda x.x = z \;\preceq^{\dagger_{\mathrm{bf}}}\; \lambda x.\mathtt{false} \qquad \text{(Freshness)}$$

$$\nu x.f \;\simeq^{\dagger_{\mathrm{bf}}}\; f \qquad \text{(Locality)}$$

$$\mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in}\ (f\ (\lambda\_.x :=\!!x + 1))\ (\lambda\_.!x)$$
$$\simeq^{\dagger_{\mathrm{bf}}}\; \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in}\ (f\ (\lambda\_.x :=\!!x - 1))\ (\lambda\_.-!x) \qquad \text{(Parametricity 1)}$$

$$\mathtt{let}\ x = \mathtt{ref}\ 1\ \mathtt{in}\ \lambda f.((f\ ());!x) \;\simeq^{\dagger_{\mathrm{bf}}}\; \lambda f.((f\ ());1) \qquad \text{(Parametricity 2)}$$

The first two equivalences originate in the $\nu$-calculus of Pitts and Stark [1993], a (call-by-value) lambda-calculus extended with construct $\nu x.t$ which creates a new *name* and binds it to $x$ in $t$, plus equality testing on names. These names are virtually identical to unit references, therefore for economy of presentation we will consider them as such. The proof techniques in *loc. cit.* rely on logical relations, which make essential use of the typing structure of the language. In contrast, we can show that these equivalences hold even in untyped state, with the caveat that one becomes a refinement, rather than an equivalence, since equality checking can get stuck if not applied to names.

The other equivalences originate in the Idealised Algol literature [O'Hearn and Tennent, 1997], a call-by-name language with ground-type *local* variables. The proofs again use logical relations, but formulated in a denotational semantics of functor categories. Escalating the proofs to call-by-value and higher-order state requires more complex techniques such as step-indexed logical relations [Ahmed et al., 2009]. Our approach will be seen to be rather different, involving a case analysis of the operations involved as they interact in all possible pairwise combinations, in a type-free setting.

## 3.5 Technical details of focussed graph rewriting

In this section we give a definition of the abstract machine introduced informally in Sec. 3.3.4. We call it a *universal abstract machine* because it can be seen as a universal algebra (the operations) combined with a mechanism for sharing or copying resources and scheduling evaluation. The machine, and hence the definitions below, are all globally parametrised by some operation set $\mathbb{O}$ and its behaviour $B_{\mathbb{O}}$.

### 3.5.1 Auxiliary definitions

We use the terms *incoming* and *outgoing* to characterise the incidence relation between neighbouring edges. Conventionally incidence is defined relative to nodes, but we find it helpful to extend this notion to edges.

**Definition 3.5.1** (Incoming and outgoing edges)**.** An *incoming* edge of an edge $e$ has a target that is a source of the edge $e$. An *outgoing* edge of the edge $e$ has a source that is a target of the edge $e$.

**Definition 3.5.2** (Path)**.** A *path* in a hypergraph is given by a non-empty sequence of edges, where an edge $e$ is followed by an edge $e'$ if the edge $e$ is an incoming edge of the edge $e'$.

Note that, in general, the first edge (resp. the last edge) of a path may have no source (resp. target). A path is said to be *from* a vertex $v$, if $v$ is a source of the first edge of the path. Similarly, a path is said to be *to* a vertex $v'$, if $v'$ is a target of the last edge of the path. A hypergraph $G$ is itself said to be a path, if all edges of $G$ comprise a path from an input (if any) and an output (if any) and every vertex is an endpoint of an edge.

**Definition 3.5.3** (Reachability)**.** A vertex $v'$ is *reachable* from a vertex $v$ if $v = v'$ holds, or there exists a path from the vertex $v$ to the vertex $v'$.

To represent SPARTAN terms, we fix the vertex label set $L$ and the edge label set $M_{\mathbb{O}}$ as described in Sec. 3.3.3, using the given operation set $\mathbb{O}$.

$$
\begin{aligned}
L &:= \{\star, \diamond\} \cup \{T^n(\star) \mid n \in \mathbb{N}\} \\
M_{\mathbb{O}} &:= \{\mathsf{I} : \star \Rightarrow \diamond,\ ? : \star \Rightarrow \star,\ \checkmark : \star \Rightarrow \star,\ \lightning : \star \Rightarrow \star\} \\
&\quad \cup \{\phi : \star \Rightarrow \star^{\otimes m_{\mathsf{e}}} \otimes \otimes_{i=1}^{m_{\mathsf{d}}} T^{n_i}(\star) \mid (\phi \vdash (m_{\mathsf{e}}; n_1, \ldots, n_{m_{\mathsf{d}}})) \in \mathbb{O}\} \\
&\quad \cup \{\circ : \diamond \Rightarrow \star,\ \otimes^{\ell}_{\mathsf{W}} : \epsilon \Rightarrow \ell,\ \otimes^{\ell}_{\mathsf{C}} : \ell^{\otimes 2} \Rightarrow \ell \mid \ell \in \{\star, \diamond\}\}
\end{aligned}
$$

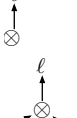
**Definition 3.5.4** (Operation path)**.** A path whose edges are all labelled with operations is called *operation path*.

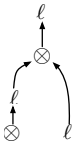
**Definition 3.5.5** (Contraction tree)**.** For each $\ell \in \{\star, \diamond\}$, a *contraction tree* is a hypernet $(C : \ell^{\otimes k} \Rightarrow \ell) \in \mathcal{H}(\{\ell\}, \{\otimes^{\ell}_{\mathsf{W}}, \otimes^{\ell}_{\mathsf{C}}\})$, such that the unique output is reachable from each vertex.

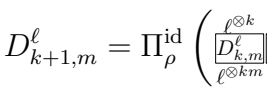
It can be observed that, for any contraction tree, an input (if any) is not an output but a source of a contraction edge.

**Definition 3.5.6** (Distributor)**.** We define a family $\{D^{\ell}_{k,m} : \ell^{\otimes km} \Rightarrow \ell^{\otimes k}\}_{k \in \mathbb{N}}$, with $\ell \in \{\star, \diamond\}$, of hypernets which we call *distributors*, inductively as follows:

$$
\begin{aligned}
D^{\ell}_{0,m} &= \emptyset \\
D^{\ell}_{1,0} &= \\
D^{\ell}_{1,1} &= \\
D^{\ell}_{1,m+2} &= \\
D^{\ell}_{k+1,m} &= \Pi^{\mathrm{id}}_{\rho}
\end{aligned}
$$

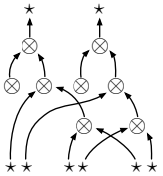
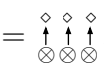




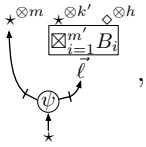


where $\emptyset$ denotes the empty hypernet, id is the identity map, and $\rho$ is a bijection such

that, for each $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, m\}$, $\rho(j + (k+1)(i-1)) = j + k(i-1)$ and $\rho((k+1)i) = km + i$.

Examples of distributors are $D^*_{2,3} =$  and $D^\diamond_{3,0} =$  . When $k = 1$, a distributor $D^\ell_{1,m}$ is a contraction tree that includes one weakening edge.

**Definition 3.5.7** (Box/stable hypernets)**.** If a hypernet is a path of only one box edge, it is called *box* hypernet. A *stable* hypernet is a hypernet $(G : \star \Rightarrow \otimes_{i=1}^m \ell_i) \in \mathcal{H}(L, \{\mathsf{I}\} \cup \mathbb{O}_\checkmark)$, such that $\otimes_{i=1}^m \ell_i \in (\{\diamond\} \cup \{T^n(\star) \mid n \in \mathbb{N}\})^m$ and each vertex is reachable from the unique input.

**Definition 3.5.8** (Copyable hypernets)**.** A hypernet $H : \star \Rightarrow \star^{\otimes k} \otimes \diamond^{\otimes h}$ is called *copyable* if it is  or  , where $\phi \in \mathbb{O}$, and each $B_i$ is a box hypernet.

**Definition 3.5.9** (One-way hypernets)**.** A hypernet $H$ is *one-way* if, for any pair $(v_i, v_o)$ of an input and an output of $H$ such that $v_i$ and $v_o$ both have type $\star$, any path from $v_i$ to $v_o$ is not an operation path.

**Remark 3.5.10** (Distributors)**.** *To the reader familiar with diagrammatic languages based on monoidal categories equipped with "sharing" (co)monoid operators, such as the ZX-calculus by Coecke and Duncan [2011], the distributors may seem an awkward alternative to quotienting the hypernets by the equational properties of the (co)monoid operator. Indeed a formulation of* Spartan *semantics in which distributors are collapsed into n-ary contractions would be quite accessible.*

*However, the structural laws of* Spartan *including equational properties of contraction, mentioned in Sec. 3.4.1, can be invalidated by certain ill-behaved but definable operations. Forcing these properties into the framework does not seem to be practically possible, as it leads to intractable interactions between such complex n-ary contractions and operations in the context as required by the key notion of* robustness *which will be introduced in Sec. 4.3.2.*

*In Sec. 4.5.2, we will introduce the equational properties of contraction that are validated by the extrinsic operations described in Sec. 3.4. These equational properties do not make contractions and weakenings form a (co)monoid, but they enable us to identify contraction trees so long as the trees contain at least one weakening. If we see the equations on contraction trees as rewrite rules from left to right, distributors are indeed normal forms with respect to these rules.*

### 3.5.2 Focussed hypernets

**Definition 3.5.11.** A token edge in a hypergraph is said to be *exposed* if its source is an input and its target is an output, and *self-acyclic* if its source and its target are different vertices.

**Definition 3.5.12** (Focussed hypernets)**.** A hypernet is said to be *focussed* if it contains only one token edge, and moreover, the token edge is shallow, self-acyclic and not exposed.

Focussed hypernets are typically ranged over by $\dot{G}, \dot{H}, \dot{N}$.

*Focus-free* hypernets are given by $\mathcal{H}_\omega(L, M_{\mathbb{O}} \backslash \{?, \checkmark, \lightning\})$, i.e. hypernets without token edges. A focussed hypernet $\dot{G}$ can be turned into an *underlying* focus-free hypernet $|\dot{G}|$ with the same type, by removing its unique token edge and identifying the source and the target of the edge. When a focussed hypernet $\dot{G}$ has a $\mathsf{t}$-token, then changing the token label $\mathsf{t}$ to another one $\mathsf{t}'$ yields a focussed hypernet denoted by $\langle \dot{G} \rangle_{\mathsf{t}'/\mathsf{t}}$. The source (resp. target) of a token is called *token source* (resp. *token target*) in short.

Given a focus-free hypernet $G$, a focussed hypernet $\mathsf{t};_i G$ with the same type can be yielded by connecting a $\mathsf{t}$-token to the $i$-th input of $G$ if the input has type $\star$. Similarly, a focussed hypernet $G;_i \mathsf{t}$ with the same type can be yielded by connecting a $\mathsf{t}$-token to the $i$-th output of $G$ if the output has type $\star$. If it is not ambiguous, we omit the index $i$ in the notation $;_i$.

### 3.5.3 Contexts

The set of *holed* hypernets (typically ranged over by $\mathcal{C}$) is given by $\mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$, where the edge label set $M_{\mathbb{O}}$ is extended by a set $\mathbb{M}$ of *hole* labels. Hole labels are typed, and typically ranged over by $\chi : \vec{\ell} \Rightarrow \vec{\ell'}$.

**Definition 3.5.13** (Contexts)**.** A holed hypernet $\mathcal{C}$ is said to be a *context* if each hole label appears at most once (at any depth) in $\mathcal{C}$.

**Definition 3.5.14.** A context is said to be *simple* if it contains a single hole, and moreover, the hole is shallow.

When $\vec{\chi}$ gives a list of all and only hole labels that appear in a context $\mathcal{C}$, the context can be also written as $\mathcal{C}[\vec{\chi}]$; a hypernet in $\mathcal{H}_\omega(L, M_{\mathbb{O}})$ can be seen as a "context without a hole", $\mathcal{C}[\,]$.

Let $\mathcal{C}[\vec{\chi^1}, \chi, \vec{\chi^2}]$ and $\mathcal{C}'[\vec{\chi^3}]$ be contexts, such that the hole $\chi$ and the latter context $\mathcal{C}'$ have the same type and $\vec{\chi^1} \cap \vec{\chi^2} \cap \vec{\chi^3} = \emptyset$. A new context $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}] \in \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \vec{\chi^1} \cup \vec{\chi^3} \cup \vec{\chi^2})$ can be obtained by *plugging* $\mathcal{C}'$ into $\mathcal{C}$: namely, by replacing the (possibly deep) hole edge of $\mathcal{C}$ that has label $\chi$ with the context $\mathcal{C}'$, and by identifying each input (resp. output) of $\mathcal{C}'$ with its corresponding source (resp. target) of the hole edge (Def. A.2.1). Each edge of the new context $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^3}]$ is inherited from either $\mathcal{C}$ or $\mathcal{C}'$, keeping the type; this implies that the new context is indeed a context with hole labels $\vec{\chi^1}, \vec{\chi^3}, \vec{\chi^2}$. Inputs and outputs of the new context coincide with those of the original context $\mathcal{C}$, and hence these two contexts have the same type. The plugging is associative in two senses: plugging two contexts into two holes of a context yields the same result regardless of the order, i.e. $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}, \mathcal{C}'', \vec{\chi^3}]$ is well-defined; and nested plugging yields the same result regardless of the order, i.e. $\mathcal{C}[\vec{\chi^1}, \mathcal{C}'[\vec{\chi^3}, \mathcal{C}'', \vec{\chi^4}], \vec{\chi^2}] = (\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}])[\vec{\chi^1}, \vec{\chi^3}, \mathcal{C}'', \vec{\chi^4}, \vec{\chi^2}]$.

The notions of focussed and focus-free hypernets can be naturally extended to contexts. In a focussed context $\dot{\mathcal{C}}[\vec{\chi}]$, the token is said to be *entering* if it is an

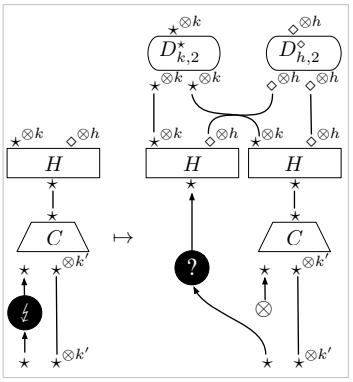



Figure 3.11: Contraction rules, with $C$ a contraction tree, and $H$ a copyable hypernet

incoming edge of a hole, and *exiting* if it is an outgoing edge of a hole. The token may be both entering and exiting.

### 3.5.4 States and transitions

Given the two parameters $\mathbb{O}$ and $B_{\mathbb{O}}$, the *universal abstract machine* $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ is defined as a state transition system. It is namely given by data $(S_{\mathbb{O}}, T \uplus B_{\mathbb{O}})$ as follows, each of which we will describe in the sequel.

- $S_{\mathbb{O}} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}})$ is a set of *states*,
- $T \subseteq S_{\mathbb{O}} \times S_{\mathbb{O}}$ is a set of *intrinsic transitions*, and
- $B_{\mathbb{O}} \subseteq S_{\mathbb{O}} \times S_{\mathbb{O}}$ is a set of *extrinsic transitions*.

A focussed hypernet of type $\star \Rightarrow \epsilon$ in $\mathcal{H}_\omega(L, M_{\mathbb{O}})$ is said to be a *state*. A state $\dot{G}$ is called *initial* if $\dot{G} = ?; |\dot{G}|$, and *final* if $\dot{G} = \checkmark; |\dot{G}|$. A state is said to be *stuck* if it is not final and cannot be followed by any transition. An *execution* on a focus-free hypernet $G : \star \Rightarrow \epsilon$ is a sequence of transitions starting from an initial state $?; G$. The following will be apparent once transitions are defined: initial states are indeed initial in the sense that no search transition results in an initial state; and final states are indeed final in the sense that no transition is possible from a final state.

The interaction rules in Fig. 3.5 specify the first class of intrinsic transitions, *search* transitions, and the contraction rules in Fig. 3.11 specify the second class

of intrinsic transitions, *copy* transitions. These intrinsic transitions are defined as follows: for each interaction rule $\dot{G} \overset{\bullet}{\mapsto} \dot{G}'$ (or resp. contraction rule $\dot{G} \overset{\otimes}{\mapsto} \dot{G}'$), if there exists a focus-free simple context $\mathcal{C}[\chi] : \star \Rightarrow \epsilon$ such that $\mathcal{C}[\dot{G}]$ and $\mathcal{C}[\dot{G}']$ are states, $\mathcal{C}[\dot{G}] \to \mathcal{C}[\dot{G}']$ is a search transition (or resp. copy transition).

Search transitions are deterministic, because at most one interaction rule can be applied at any state. Although two different contraction rules may be possible at a state, copy transitions are still deterministic. Namely, if two different contraction rules $\dot{G} \mapsto \dot{G}'$ and $\dot{H} \mapsto \dot{H}'$ can be applied to the same state, i.e. there exist focus-free simple contexts $\mathcal{C}_G$ and $\mathcal{C}_H$ such that $\mathcal{C}_G[\dot{G}] = \mathcal{C}_H[\dot{H}]$, then these two rules yield the same transition, by satisfying $\mathcal{C}_G[\dot{G}'] = \mathcal{C}_H[\dot{H}']$. Informally, in Fig. 3.11, $H$ is determined uniquely and the choice of $C$ does not affect the result.

Intrinsic transitions are therefore all deterministic, and moreover, search transitions are reversible because the inverse of the interaction rules is again deterministic. When a sequence $\dot{G} \to^* \dot{G}'$ of transitions consists of search transitions only, it is annotated by the symbol $\bullet$ as $\dot{G} \overset{\bullet}{\to}^* \dot{G}'$.

An execution on any stable net, or on representation of any value, terminates successfully at a final state with only search transitions (by Lem. A.4.2, Lem. A.4.4 and Lem. A.4.6(1)).

The behaviour $B_{\mathbb{O}}$, which is a parameter of the machine, specifies a set of extrinsic transitions. Extrinsic transitions are also called *compute* transitions, and each of them must target an active operation. Namely, a transition $\dot{G} \to \dot{G}'$ is a compute transition if: the first state $\dot{G}$ has a rewrite token ($\lightning$) that is an incoming edge of an active operation edge; and the second state $\dot{G}'$ has a search token (?). Copy transitions or compute transitions are possible if and only if a state has a rewrite token ($\lightning$), and they always change the token to a search token (?). We refer to copy transitions and compute transitions altogether as *rewrite* transitions.

Compute transitions may be specified locally, by *rewrite rules*, in the same manner as the intrinsic transitions. The rewrite rules introduced in Sec. 3.4.2 are such

examples. However, we leave it entirely open what the actual rewrite associated to some operation is, by having the behaviour $B_{\mathbb{O}}$ as parameter as well as the operation set $\mathbb{O}$. This is part of the semantic flexibility of our framework. We do not specify a meta-language for encoding effects as particular transitions. Any *algorithmic* state transformation is acceptable.

# Chapter 4

# Robustness and observational equivalence

## 4.1 Outline

This chapter presents a novel proof methodology of observational equivalence, offered by the UAM. In focussed graph rewriting that is performed by the UAM, information of program-execution status is centralised to the token (or *focus*), and each step of program execution is determined by the token and its neighbourhood. This enables a new style of reasoning centred around a graph-theoretic intuition of *locality*, in which one can analyse how a program fragment evolves during program execution by examining how the token interacts with the fragment.

Exploiting local reasoning yields a case-by-case reasoning principle for proving observational equivalence between two program fragments $t$ and $u$. The proof namely boils down to establishing coincidence between the way these two fragments interact with the token. This can be done by direct, step-wise, comparison between two executions of programs $C[t]$ and $C[u]$, the fragments in an arbitrary common context $C$. At each step of these executions, we can enumerate possible interaction between the token and either of the fragments, and identify sufficient conditions for the fragments to have the same interaction with the token. A key sufficient condition,

*robustness*, is our conceptual contribution. It characterises when the fragments are respected by a rewrite triggered by the token.

The main technical result, a *characterisation theorem* (Thm. 4.3.14), formalises this local reasoning principle for proving observational equivalence, and identifies the sufficient conditions including robustness. The theorem focuses on the UAM that is deterministic, to avoid over-complication of the technical development. Although this restriction leaves some computational effects beyond the scope, the deterministic UAM can still accommodate interesting effects such as state and exception. We will illustrate that the theorem can be used to prove some challenging observational equivalences from the literature, involving arbitrary (untyped) state.

Additionally, we propose a generalised notion of observational equivalence that has two parameters: a class of contexts and a preorder on natural numbers. The first parameter enables us to quantify over some contexts, instead of all contexts as in the standard notion. This can be used to identify a shape of contexts that respects or violates certain observational equivalences, given that not necessarily all arbitrarily generated contexts arise in program execution. The second parameter, a preorder on natural numbers, deals with numbers of steps it takes for the UAM to terminate. Taking the universal relation recovers the standard notion of observational equivalence. Observational equivalence with respect to the greater-than-equal relation, for example, means that replacing a fragment with another in any programs (within the class specified by the first parameter) never increases the number of execution steps.

This chapter is organised as follows. The generalised notion of observational equivalence is defined in Sec. 4.2, and the characterisation theorem is presented in Sec. 4.3. The proof of the theorem is given in Sec. 4.4, some of whose details can be found in an appendix. Sec. 4.5 gives applications of the characterisation theorem to proving observational equivalence.

## 4.2 Contextual refinement and equivalence

We propose notions of contextual refinement and equivalence that check for successful termination of execution. These notions generalise the standard notions, by additionally taking into account a class of contexts to quantify over, and also the number of transitions. They are namely with respect to the universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ with some operation set $\mathbb{O}$ and its behaviour $B_{\mathbb{O}}$, and parametrised by the following: a set $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$ of focus-free contexts that is closed under plugging (i.e. for any contexts $\mathcal{C}[\vec{\chi^1}, \chi, \vec{\chi^2}], \mathcal{C}' \in \mathbb{C}$ such that $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}]$ is defined, $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}] \in \mathbb{C}$); and a preorder $Q$ on natural numbers.

**Definition 4.2.1** (State refinement and equivalence)**.** Let $Q$ be a preorder on $\mathbb{N}$, and $\dot{G}_1$ and $\dot{G}_2$ be two states.

- $\dot{G}_1$ is said to *refine* $\dot{G}_2$ up to $Q$, written as $B_{\mathbb{O}} \models (\dot{G}_1 \mathrel{\dot{\preceq}_Q} \dot{G}_2)$, if for any number $k_1 \in \mathbb{N}$ and any final state $\dot{N}_1$ such that $\dot{G}_1 \rightarrow^{k_1} \dot{N}_1$, there exist a number $k_2 \in \mathbb{N}$ and a final state $\dot{N}_2$ such that $k_1 \; Q \; k_2$ and $\dot{G}_2 \rightarrow^{k_2} \dot{N}_2$.

- $\dot{G}_1$ and $\dot{G}_2$ are said to be *equivalent* up to $Q$, written as $B_{\mathbb{O}} \models (\dot{G}_1 \mathrel{\dot{\simeq}_Q} \dot{G}_2)$, if $B_{\mathbb{O}} \models (\dot{G}_1 \mathrel{\dot{\preceq}_Q} \dot{G}_2)$ and $B_{\mathbb{O}} \models (\dot{G}_2 \mathrel{\dot{\preceq}_Q} \dot{G}_1)$.

**Definition 4.2.2** (Contextual refinement and equivalence)**.** Let $\mathbb{C}$ be a set of contexts that is closed under plugging, $Q$ be a preorder on $\mathbb{N}$, and $H_1$ and $H_2$ be focus-free hypernets of the same type.

- $H_1$ is said to *contextually refine* $H_2$ in $\mathbb{C}$ up to $Q$, written as $B_{\mathbb{O}} \models (H_1 \preceq_Q^{\mathbb{C}} H_2)$, if any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$, such that $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are states, yields refinement $B_{\mathbb{O}} \models (?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_2])$.

- $H_1$ and $H_2$ are said to be *contextually equivalent* in $\mathbb{C}$ up to $Q$, written as $B_{\mathbb{O}} \models (H_1 \simeq_Q^{\mathbb{C}} H_2)$, if $B_{\mathbb{O}} \models (H_1 \preceq_Q^{\mathbb{C}} H_2)$ and $B_{\mathbb{O}} \models (H_2 \preceq_Q^{\mathbb{C}} H_1)$.

In the sequel, we simply write $\dot{G}_1 \dot{\preceq}_Q \dot{G}_2$ etc., making the parameter $B_{\mathbb{O}}$ implicit.

Because $Q$ is a preorder, $\dot{\preceq}_Q$ and $\preceq_Q^{\mathbb{C}}$ are indeed preorders, and accordingly, equivalences $\dot{\simeq}_Q$ and $\simeq_Q^{\mathbb{C}}$ are indeed equivalences (Lem. A.5.2). Examples of preorder $Q$ include: the universal relation $\mathbb{N} \times \mathbb{N}$, the "greater-than-or-equal" order $\geq_{\mathbb{N}}$, and the equality $=_{\mathbb{N}}$.

When the relation $Q$ is the universal relation $\mathbb{N}\times\mathbb{N}$, the notions concern successful termination, and the number of transitions is irrelevant. If all compute transitions are deterministic, contextual equivalences $\simeq_{\geq_{\mathbb{N}}}^{\mathbb{C}}$ and $\simeq_{=_{\mathbb{N}}}^{\mathbb{C}}$ coincide for any $\mathbb{C}$ (as a consequence of Lem. A.5.3).

Because $\mathbb{C}$ is closed under plugging, the contextual notions $\preceq_Q^{\mathbb{C}}$ and $\simeq_Q^{\mathbb{C}}$ indeed become congruences. Namely, for any $H_1 \,\Box^{\mathbb{C}}\, H_2$ and $\mathcal{C} \in \mathbb{C}$ such that $\mathcal{C}[H_1]$ and $\mathcal{C}[H_2]$ are defined, $\mathcal{C}[H_1] \,\Box^{\mathbb{C}}\, \mathcal{C}[H_2]$, where $\Box \in \{\preceq_Q, \simeq_Q\}$.

As the parameter $\mathbb{C}$, we will particularly use the set $\mathbb{C}_{\mathbb{O}} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$ of any focus-free contexts, and its subset $\mathbb{C}_{\mathbb{O}\text{-bf}}$ of *binding-free* contexts.

**Definition 4.2.3** (Binding-free contexts)**.** A focus-free context $\mathcal{C}$ is said to be *binding-free* if there exists no path, at any depth, from a source of a contraction, atom, box or hole edge, to a source of a hole edge.

The set $\mathbb{C}_{\mathbb{O}}$ is closed under plugging, and so is the set $\mathbb{C}_{\mathbb{O}\text{-bf}}$ (Lem. A.5.1). Restriction to binding-free contexts is useful to focus on call-by-value languages, because only values will be bound during evaluation in these languages. The restriction syntactically means forbidding the hole of contexts from appearing in the bound positions, as discussed below.

The standard notions of contextual refinement and equivalence can be recovered as $\preceq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}}}$ and $\simeq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}}}$, by taking the set $\mathbb{C}_{\mathbb{O}} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}}\cup\mathbb{M})$ of all focus-free contexts, and the universal relation $\mathbb{N} \times \mathbb{N}$.

### 4.2.1 Observational equivalences on terms

The notion of observational refinement on terms, informally introduced in Sec. 3.4, can now be defined using the contextual refinement on hypernets as follows. Recall that the observational refinement is parametrised by an operation set $\mathbb{O}$ and its behaviour $B_{\mathbb{O}}$. Given two derivable judgements $\vec{x} \mid \vec{a} \vdash t_1 : \star$ and $\vec{x} \mid \vec{a} \vdash t_2 : \star$, we write:

$$
\begin{array}{lll}
B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger_{\mathrm{all}}} t_2 : \star) & \text{if} & B_{\mathbb{O}} \models ((\vec{x} \mid \vec{a} \vdash t_1 : \tau)^{\dagger} \preceq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}}} (\vec{x} \mid \vec{a} \vdash t_2 : \tau)^{\dagger}), \\
B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger_{\mathrm{all}}} t_2 : \star) & \text{if} & B_{\mathbb{O}} \models ((\vec{x} \mid \vec{a} \vdash t_1 : \tau)^{\dagger} \simeq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}}} (\vec{x} \mid \vec{a} \vdash t_2 : \tau)^{\dagger}), \\
B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger_{\mathrm{bf}}} t_2 : \star) & \text{if} & B_{\mathbb{O}} \models ((\vec{x} \mid \vec{a} \vdash t_1 : \tau)^{\dagger} \preceq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}\text{-bf}}} (\vec{x} \mid \vec{a} \vdash t_2 : \tau)^{\dagger}), \\
B_{\mathbb{O}} \models (\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger_{\mathrm{bf}}} t_2 : \star) & \text{if} & B_{\mathbb{O}} \models ((\vec{x} \mid \vec{a} \vdash t_1 : \tau)^{\dagger} \simeq_{\mathbb{N}\times\mathbb{N}}^{\mathbb{C}_{\mathbb{O}\text{-bf}}} (\vec{x} \mid \vec{a} \vdash t_2 : \tau)^{\dagger}).
\end{array}
$$

The refinements $\preceq^{\dagger_{\mathrm{all}}}$ and $\preceq^{\dagger_{\mathrm{bf}}}$ enjoy different congruence properties, as specified by the set $\mathbb{C}_{\mathbb{O}}$ of focus-free contexts (as hypernets) and its binding-free restriction $\mathbb{C}_{\mathbb{O}\text{-bf}}$. This difference can also be described in terms of syntactical contexts as follows. Let *term-contexts* and their *binding-free* restriction be defined by the following grammar:

$$
\begin{array}{rll}
C ::= [\,] \mid \texttt{new}\ a \multimap t\ \texttt{in}\ C \mid \texttt{bind}\ x \to t\ \texttt{in}\ C & & \\
\quad \mid \texttt{new}\ a \multimap C\ \texttt{in}\ t \mid \texttt{bind}\ x \to C\ \texttt{in}\ t & & \\
\quad \mid \vec{y}.C \mid \phi(\vec{t}, C, \vec{t'}; \vec{s}) \mid \phi(\vec{t}; \vec{s}, C, \vec{s'}) & & \text{(term-contexts)} \\
\tilde{C} ::= [\,] \mid \texttt{new}\ a \multimap t\ \texttt{in}\ \tilde{C} \mid \texttt{bind}\ x \to t\ \texttt{in}\ \tilde{C} & & \\
\quad \mid \vec{y}.\tilde{C} \mid \phi(\vec{t}, \tilde{C}, \vec{t'}; \vec{s}) \mid \phi(\vec{t}; \vec{s}, \tilde{C}, \vec{s'}) & & \text{(binding-free term-contexts)}
\end{array}
$$

The type system of SPARTAN (Fig. 3.2) can be extended to term-contexts, by annotating the hole '$[\,]$' as '$[\,]_{\vec{x}\mid\vec{a}}$' and adding a typing rule $\overline{\vec{x} \mid \vec{a} \vdash [\,]_{\vec{x}\mid\vec{a}} : \star}$. We write $C[\,]_{\vec{x}\mid\vec{a}}$ when the hole of $C$ is annotated with $\vec{x} \mid \vec{a}$.

**Lemma 4.2.4.** *Let $\vec{x} \mid \vec{a} \vdash t_1 : \star$ and $\vec{x} \mid \vec{a} \vdash t_2 : \star$ be derivable judgements. Let $C$ be a term-context, $\vec{x'}$ be a sequence of variables and $\vec{a'}$ be a sequence of atoms, such that $\vec{x'} \mid \vec{a'} \vdash C[\,]_{\vec{x}|\vec{a}} : \star$.*

1. *If $\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger}_{\mathrm{all}} t_2 : \star$, then $\vec{x'} \mid \vec{a'} \vdash C[t_1] \preceq^{\dagger}_{\mathrm{all}} C[t_2] : \star$.*
2. *If $\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger}_{\mathrm{all}} t_2 : \star$, then $\vec{x'} \mid \vec{a'} \vdash C[t_1] \simeq^{\dagger}_{\mathrm{all}} C[t_2] : \star$.*
3. *If $\vec{x} \mid \vec{a} \vdash t_1 \preceq^{\dagger}_{\mathrm{bf}} t_2 : \star$, and $C$ is binding-free, then $\vec{x'} \mid \vec{a'} \vdash C[t_1] \preceq^{\dagger}_{\mathrm{bf}} C[t_2] : \star$.*
4. *If $\vec{x} \mid \vec{a} \vdash t_1 \simeq^{\dagger}_{\mathrm{bf}} t_2 : \star$, and $C$ is binding-free, then $\vec{x'} \mid \vec{a'} \vdash C[t_1] \simeq^{\dagger}_{\mathrm{bf}} C[t_2] : \star$.*

*Proof outline.* The translation $(-)^\dagger$ of Spartan terms to hypernets (Fig. 3.4) can be extended to term-contexts, by translating the rule $\overline{\vec{x} \mid \vec{a} \vdash [\,]_{\vec{x}|\vec{a}} : \star}$ into a path hypernet $\chi : \star \Rightarrow \star^{\otimes|\vec{x}|} \otimes \diamond^{\otimes|\vec{a}|}$. Translating term-contexts indeed yields (graphical) contexts, and translating binding-free term-contexts yields (graphical) binding-free contexts (proof by induction on (binding-free) term-contexts).

For each $i \in \{1, 2\}$, a judgement $\vec{x'} \mid \vec{a'} \vdash C[t_i] : \star$ is derivable, and moreover,

$$(\vec{x'} \mid \vec{a'} \vdash C[t_i] : \star)^\dagger = (\vec{x'} \mid \vec{a'} \vdash C[\,]_{\vec{x}|\vec{a}} : \star)^\dagger[(\vec{x} \mid \vec{a} \vdash t_i : \star)^\dagger]$$

(proof by induction on $C$). The congruence property of contextual refinements concludes the proof. □

## 4.3 A characterisation theorem

We can now formalise a proof method for contextual refinement and equivalence. All the technical development in this section is with respect to the universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$, parametrised by an operation set $\mathbb{O}$ and its behaviour $B_{\mathbb{O}}$, that satisfies some conditions including determinism.

Firstly in Sec. 4.3.1, we state the conditions of the machine, to which the proof method applies. Sec. 4.3.2 formalises the proof method as a *characterisation theorem*

(Thm. 4.3.14), introducing the key concept of *robustness*. Additionally, in Sec. 4.3.3, we list some useful lemmas that can be used in robustness check.

### 4.3.1 Determinism and refocusing

We focus on the situation where the universal abstract machine is both *deterministic* as a state transition system, and *refocusing* in the following sense.

**Definition 4.3.1** (Rooted states and refocusing machine)**.**

- A state $\dot{G}$ is *rooted* if $?;|\dot{G}| \xrightarrow{\bullet}{}^{*} \dot{G}$.
- The universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ is *refocusing* if every transition preserves the rooted property.

Recall that intrinsic transitions are all deterministic, and that intrinsic transitions and extrinsic transitions are mutually exclusive. The machine becomes deterministic, as a state transition system, if compute transitions specified by the behaviour $B_{\mathbb{O}}$ are deterministic.

Any initial state is trivially rooted, and search transitions preserve the rooted property. The *stationary* property below gives a sufficient condition for a rewrite transition to preserve the rooted property (Lem. A.3.8).

**Definition 4.3.2** (Stationary rewrite transitions)**.** A rewrite transition $\dot{G} \to \dot{G}'$ is *stationary* if there exist a focus-free simple context $\mathcal{C}$, focus-free hypernets $H$ and $H'$, and a number $i \in \mathbb{N}$, such that $H$ is one-way, $\dot{G} = \mathcal{C}[\not{\iota}\,;_i H]$ and $\dot{G}' = \mathcal{C}[?;_i H']$, and the following holds. For any $j \in \mathbb{N}\backslash\{i\}$, such that $\mathcal{C}[?;_j H]$ is a state, there exists a state $\dot{N}$ with a rewrite token, such that $\mathcal{C}[?;_j H] \xrightarrow{\bullet} \dot{N}$.

Copy transitions are stationary, and hence they preserve the rooted property, because each input of the contraction tree $C$ in Fig. 3.11 is a source of a contraction edge. Therefore, the machine becomes refocusing if the behaviour $B_{\mathbb{O}}$ specifies compute transitions that preserve the rooted property.

**Remark 4.3.3** (Refocusing)**.** *When a rewrite transition results in a rooted state $\dot{N}'$, starting the search process (i.e. search transitions) from the state $\dot{N}'$ can be seen as* resuming *the search process $?;|\dot{N}'| \xrightarrow{\bullet}{}^{*} \dot{N}'$, from an initial state, on the underlying hypernet $|\dot{N}'|$. Resuming redex search after a rewrite, rather than starting from scratch, is an important aspect of abstract machines. In the case of the lambda-calculus, enabling the resumption is identified as one of the key steps (called* refocusing*) to synthesise abstract machines from reduction semantics by Danvy et al. [2012]. In our setting, it is preservation of the rooted property that justifies the resumption.*

*The stationary property, as a sufficient condition of preservation of the rooted property, characterises many operations with local behaviour. Compute transitions of operations that involve non-local change of a token position, like label jumping, could preserve the rooted property without being stationary.*

### 4.3.2 Templates and robustness

A candidate for contextual refinement, called *pre-template*, is given by a family of pairs of hypernets, indexed by types.

**Definition 4.3.4** (Pre-template)**.** A *pre-template* is a union $\lhd := \cup_{I \in \mathcal{I}} \lhd_I$ of a family of binary relations on focus-free hypernets $\mathcal{H}_\omega(L, M_{\mathbb{O}} \backslash \{?, \checkmark, \lightning\})$ indexed by a set $\mathcal{I}$ of types, such that for any $G_1 \lhd_I G_2$ where $I \in \mathcal{I}$, $G_1$ and $G_2$ are focus-free hypernets with type $G_1 : I$ and $G_2 : I$.

Obviously, if $\lhd$ is a pre-template, its converse $\lhd^{-1}$ is also a pre-template.

Pre-templates do not necessarily relate hypernets that represent terms, nor hypernets that arise from rewrite rules of the operations. However, the rewrite rules are indeed natural candidates for contextual refinements, and therefore natural sources of pre-templates.

**Example 4.3.5** (Beta pre-template)**.** *As a leading example, we consider the* beta *pre-template* $\lhd^{\vec{@}}$ *derived from the beta rewrite rule (Fig. 3.7), by forgetting the token: namely,* $|\dot{G}_1| \lhd^{\vec{@}} |\dot{G}_2|$ *if* $\dot{G}_1 \mapsto \dot{G}_2$ *is a beta rewrite rule. These hypernets* $|G_1|$ *and* $|G_2|$ *have the same type* $\star \Rightarrow \star^{\otimes k} \otimes \diamond^{\otimes h} \otimes \vec{\ell}$*, where* $k, h \in \mathbb{N}$ *and the sequence* $\vec{\ell}$ *of types can be arbitrary.*

Throughout this section, let $\mathbb{C}$ be a set of contexts, and $Q$, $Q'$ and $Q''$ be binary relations on $\mathbb{N}$. Given a pre-template $\lhd$, our goal is to prove that $H_1 \lhd H_2$ implies contextual refinement $H_1 \preceq^{\mathbb{C}}_{Q} H_2$, possibly with the help of state refinement $\dot{\preceq}$ up to $Q'$ or $Q''$. As will be apparent in Sec. 4.5, the use of state refinement is particularly convenient in reasoning, allowing us to identify different contraction trees of the same type.

At the core of the proof is comparison between hypernets related by the pre-template $\lhd$, in any possible contexts specified by the set $\mathbb{C}$. In other words, the comparison is between a pair of states whose only difference is given by the pre-template $\lhd$. This pair is given by data called *specimen*. The comparison can be relaxed by allowing state refinements $\dot{\preceq}_{Q'}$ and $\dot{\preceq}_{Q''}$, in addition to the pre-template $\lhd$, to specify the difference between states. The relaxed comparison is targeted at a pair of states, which is intuitively a specimen up to state refinements, called *quasi-specimen*.

**Definition 4.3.6** ((Quasi-)specimens)**.** Let $\lhd$ be a pre-template, and $R$ and $R'$ be binary relations on states.

1. A triple $(\dot{\mathcal{C}}[\vec{\chi}]; \vec{H}^1; \vec{H}^2)$ is a $\mathbb{C}$-*specimen* of $\lhd$ if the following hold:

   (A) $|\dot{\mathcal{C}}[\vec{\chi}]| \in \mathbb{C}$, and $|\vec{\chi}| = |\vec{H}^1| = |\vec{H}^2|$.

   (B) $H^1_i \lhd H^2_i$ for each $i \in \{1, \ldots, |\vec{\chi}|\}$.

   (C) $\dot{\mathcal{C}}[\vec{H}^p]$ is a state for each $p \in \{1, 2\}$.

2. A pair $(\dot{N}_1, \dot{N}_2)$ of states is a *quasi*-$\mathbb{C}$-*specimen* of $\lhd$ *up to* $(R, R')$, if there

exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ of $\lhd$ such that the following hold:

(A) The tokens of $\dot{\mathcal{C}}$, $\dot{N}_1$ and $\dot{N}_2$ all have the same label.

(B) If $\dot{N}_1$ and $\dot{N}_2$ are rooted, then $\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$ are also rooted, $\dot{N}_1 \; R \; \dot{\mathcal{C}}[\vec{H}^1]$, and $\dot{\mathcal{C}}[\vec{H}^2] \; R' \; \dot{N}_2$.

We can refer to *the* token label of a $\mathbb{C}$-specimen and a quasi-$\mathbb{C}$-specimen. Any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ gives a quasi-$\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\vec{H}^1], \dot{\mathcal{C}}[H^2])$ up to $(=, =)$. Note that the focussed context of each $\mathbb{C}$-specimen may have multiple holes. We say a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\vec{\chi}]; \vec{H}^1; \vec{H}^2)$ is *single* if $|\vec{\chi}| = 1$, i.e. the context $\dot{\mathcal{C}}$ has exactly one hole edge (at any depth).

For the pre-template $\lhd$ to imply contextual refinement, hypernets related by $\lhd$ should intuitively induce the same behaviour of the token, regardless of contexts. We use specimens to analyse the token behaviour, in a case-by-case manner. Namely, given any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ of a pre-template $\lhd$, we analyse how the token in the context $\dot{\mathcal{C}}$ interacts with the hypernets $\vec{H}^1$, and $\vec{H}^2$, that are contributed by $\lhd$. According to the actual position and the label of the token, the possible interaction can be classified as follows:

**Case I: move inside the common context.** The token is a search token (?) or a value token (✓), and it interacts with an edge of the context $\dot{\mathcal{C}}$. It does not interact with the fragments at all.

**Case II: visit to the fragments.** The token is a search token (?) or a value token (✓), and it is set to interact with edges of the fragments. The token is necessarily next to a hole edge in the context $\dot{\mathcal{C}}$.

**Case III: rewrite.** The token in the context $\dot{\mathcal{C}}$ is a rewrite token (↯). It triggers a compute transition, in which a part of the fragments may be rewritten.

**Case IV: termination.** The token is a value token (✓) and does not have an edge to interact, which is an incoming edge of the token. In this case, both states

$\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$ are necessarily final.

Our objective is to establish that the token always interacts with the hypernets $\vec{H}^1$ in the same manner as with the hypernets $\vec{H}^2$. By step-wise reasoning whose details will appear in Sec. 4.4,it suffices to show that the interaction in the $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$, described above, always results in another (quasi-)$\mathbb{C}$-specimen of $\lhd$, unless it results in stuck states or final states.

Apart from Case IV which actually involves no interaction, it is only Case I that always results in another $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H}^1; \vec{H}^2)$ with a different context $\dot{\mathcal{C}}'$. For the other cases, i.e. Case II and Case III, we identify sufficient conditions for the pre-template $\lhd$, as well as for the operations $\mathbb{O}$, to yield a (quasi-)$\mathbb{C}$-specimen as a result.

There are two sufficient conditions for Case II, according to the token: *input-safety* for a search token (?), and *output-closure* for a value token (✓). Input-safety enumerates some situations of interaction that yield a (quasi-)specimen or a pair of stuck states. On the other hand, output-closure characterises a situation where a value token cannot interact with the hypernets contributed by the pre-template $\lhd$, under the assumption that the machine is refocusing.

**Definition 4.3.7** (Input-safety)**.** A pre-template $\lhd$ is $(\mathbb{C}, Q, Q')$-*input-safe* if, for any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ of $\lhd$ such that $\dot{\mathcal{C}}$ has an entering search token, one of the following holds.

(I) There exist two stuck states $\dot{N}_1$ and $\dot{N}_2$ such that $\dot{\mathcal{C}}[\vec{H}^p] \rightarrow^* \dot{N}_p$ for each $p \in \{1, 2\}$.

(II) There exist a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ and two numbers $k_1, k_2 \in \mathbb{N}$, such that the token of $\dot{\mathcal{C}}'$ is not a rewrite token and not entering, $(1{+}k_1)\ Q\ k_2$, $\dot{\mathcal{C}}[\vec{H}^1] \rightarrow^{1+k_1} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}[\vec{H}^2] \rightarrow^{k_2} \dot{\mathcal{C}}'[\vec{H'}^2]$.

(III) There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_1, \dot{N}_2)$ of $\lhd$ up to $(\dot{\simeq}_{Q'}, \dot{\simeq}_{Q'})$, whose token is not a rewrite token, and two numbers $k_1, k_2 \in \mathbb{N}$, such that $(1 + k_1)\ Q\ (1 + k_2)$, $\dot{\mathcal{C}}[\vec{H}^1] \rightarrow^{1+k_1} \dot{N}_1$, and $\dot{\mathcal{C}}[\vec{H}^2] \rightarrow^{1+k_2} \dot{N}_2$.

**Definition 4.3.8** (Output-closure)**.** A pre-template $\lhd$ is *output-closed* if, for any hypernets $H_1 \lhd H_2$, either $H_1$ or $H_2$ is one-way.

**Definition 4.3.9** (Templates)**.** A pre-template $\lhd$ is a $(\mathbb{C}, Q, Q')$-*template*, if it is $(\mathbb{C}, Q, Q')$-input-safe and also output-closed.

It is possible to allow a value token to interact with the hypernets contributed by the pre-template $\lhd$, and define a counterpart of input-safety for a value token. However, we here opt for a simple sufficient condition, output-closure, that excludes interaction in the refocusing machine. It is simple yet powerful enough to prove many interesting contextual refinements, as we will see in Sec. 4.5.

**Example 4.3.10** (Beta template)**.** *We continue with the beta pre-template* $\lhd^{\vec{@}}$ *from Ex. 4.3.5. It is natural to expect that the beta pre-template* $\lhd^{\vec{@}}$ *is input-safe, because it is derived from the beta rewrite rule. Given hypernets* $H_1 \lhd^{\vec{@}} H_2$*, whenever a search token enters* $H_1$*, it should eventually become a rewrite token and trigger the beta rewrite of* $H_1$ *to* $H_2$*. If this is the case, Def. 4.3.7(II) is fulfilled, where* $k_2 = 0$ *and the new context* $\dot{\mathcal{C}}'$ *has one less hole than the original context* $\dot{\mathcal{C}}$*.*

*However, the actual behaviour of an entering search token is not necessarily as expected. When a search token visits the hypernet* $G_S$ *that is the second argument of the application operation (see Fig. 3.7), there is no guarantee that the token returns to the application edge (@) and hence triggers the beta rewrite of the hypernet* $H_1$*. Even if it returns, it may have triggered some rewrites that change the hypernet* $G_S$ *to something else. Consequently, the pre-template* $\lhd^{\vec{@}}$ *is not necessarily input-safe as it is.*

*One possible solution is to restrict the pre-template* $\lhd^{\vec{@}}$*, by requiring the hypernet* $G_S$ *to be stable. This restriction ensures that a search token that enters* $H_1$ *eventually triggers the beta rewrite of* $H_1$ *to* $H_2$*, because a search token that visits the stable hypernet* $G_S$ *is guaranteed to return without triggering any rewrites (by Lem. A.4.4 and Lem. A.4.6(1)). Moreover, this restriction makes the pre-template* $\lhd^{\vec{@}}$ *output-*

*closed, because of the type of $G_S$. The restriction in fact corresponds to restriction of the standard beta law to the call-by-value one $(\lambda x.t)\ v \simeq v[t/x]$, where the argument $v$ is required to be a value.*

For Case III, where the token in the context triggers a rewrite, we identify a sufficient condition called *robustness*. It is in a similar style as input-safety, but additionally relative to the triggered rewrite, whose target is a contraction $\otimes$ or an active operation $\phi_i$.

**Definition 4.3.11** (Robustness)**.** A pre-template $\lhd$ is $(\mathbb{C}, Q, Q', Q'')$*-robust relative to* a rewrite transition $\dot{N} \to \dot{N}'$ if, for any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ of $\lhd$, such that $\dot{\mathcal{C}}[\vec{H}^1] = \dot{N}$ and the token of $\dot{\mathcal{C}}$ is a rewrite token and not entering, one of the following holds.

(I) $\dot{\mathcal{C}}[\vec{H}^1]$ or $\dot{\mathcal{C}}[\vec{H}^2]$ is not rooted.

(II) There exists a stuck state $\dot{N}''$ such that $\dot{N}' \to^* \dot{N}''$.

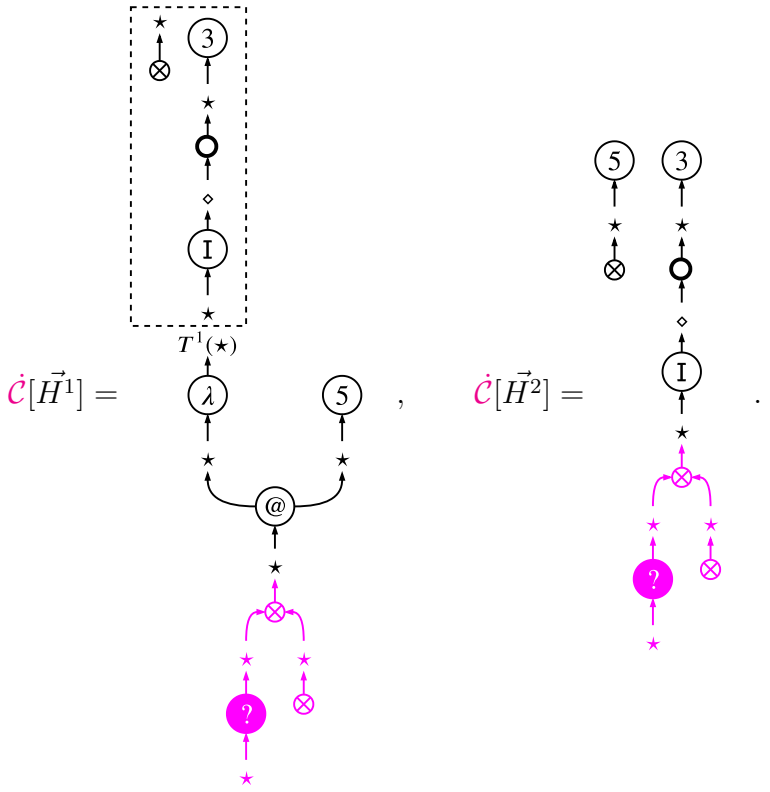
(III) There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_1'', \dot{N}_2'')$ of $\lhd$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, whose token is not a rewrite token, and two numbers $k_1, k_2 \in \mathbb{N}$, such that $(1+k_1)\ Q\ k_2$, $\dot{N}' \to^{k_1} \dot{N}_1''$, and $\dot{\mathcal{C}}[\vec{H}^2] \to^{k_2} \dot{N}_2''$.

**Example 4.3.12** (Robustness of beta pre-template)**.** *Let us see informally how robust the beta pre-template $\lhd^{\vec{@}}$, from Ex. 4.3.5, can be.*

*The beta pre-template is robust relative to the arithmetic rewrite rule in Fig. 3.8, because application of the rewrite rule does not interfere with any hypernets contributed by the pre-template $\lhd^{\vec{@}}$. Any specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ is therefore turned into another specimen $(\dot{\mathcal{C}}'; \vec{H}^1; \vec{H}^2)$ with a different context $\dot{\mathcal{C}}'$.*

*The beta pre-template is also robust relative to the beta rewrite rule. Hypernets contributed by the pre-template $\lhd^{\vec{@}}$ may appear as a part of the redex of the rewrite rule, namely, as a part of $G$ and $G_S$ in Fig. 3.7. However, the rewrite does not manipulate what is inside $G$ and $G_S$, and hence it preserves the contributed hypernets. Any specimen is again turned into another specimen with a different context.*

*When it comes to a copy transition, which applies a contraction rule (Fig. 3.11), robustness is not guaranteed in general. Starting from a pair of states given by a specimen* $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$*, it may be the case that some copy transitions are possible without reaching another (quasi-)specimen. An example scenario is when the specimen yields the following two states, where the context* $\dot{\mathcal{C}}$ *is indicated by magenta:*




*Transitions from the state* $\dot{\mathcal{C}}[\vec{H}^1]$ *eventually duplicate the application edge (@), the abstraction edge (λ) and also the entire box connected to the abstraction edge. In particular, these transitions duplicate the atom edge contained in the box. However, transitions from the state* $\dot{\mathcal{C}}[\vec{H}^2]$ *can never duplicate the atom edge, because the edge is shallow in this state. This mismatch of duplication prevents the beta pre-template from being robust relative to a copy transition.*

*This is why we might prefer to restrict contexts to be binding-free. If the context* $|\dot{\mathcal{C}}|$ *is binding-free, the situation explained above would never happen. Application of a contraction rule can involve the hypernets* $\vec{H}^1$ *and* $\vec{H}^2$ *only as a part of box con-*

*tents that are duplicated as a whole. Any specimen is therefore turned into another specimen whose context possibly has more holes, by a single copy transition. Note that this explains why we allow the context of a specimen to have multiple holes.*

*Another interesting example is given by the operation* `stat` *mentioned earlier, whose rewrite transition measures the size of the whole hypernet. The transition introduces an edge labelled by an integer, which can be seen as a passive operation with no arguments, that represents the measured size. Consequently, the rewrite transition on states* $\dot{\mathcal{C}}[\vec{H}^1]$ *and* $\dot{\mathcal{C}}[\vec{H}^2]$ *may introduce different integer edges, which cannot be related by the pre-template* $\vec{\lhd}^{\textcircled{a}}$*. Robustness of the beta pre-template fails relative to the operation* `stat`*.*

*Since robustness is only a sufficient condition, this failure does not necessarily mean that the beta pre-template cannot imply contextual refinement. Nevertheless, the failure suggests how the operation* `stat` *could violate the contextual refinement. It namely indicates that such a violation would rely on equality testing of integers, such as an active operation* $\texttt{test}(t, u; -)$ *whose rewrite transition is only defined when the two eager arguments are the same integer.*

Finally, the characterisation theorem (Thm. 4.3.14 below) enables us to prove contextual refinement $\preceq_Q^{\mathbb{C}}$ using state refinements $\dot{\preceq}_{Q'}$ and $\dot{\preceq}_{Q''}$, under the assumption that the machine is deterministic and refocusing, and that the triple $(Q, Q', Q'')$ is *reasonable* in the following sense.

**Definition 4.3.13** (Reasonable triples)**.** A triple $(Q, Q', Q'')$ of preorders on $\mathbb{N}$ is *reasonable* if the following hold:

(A) $Q$ is closed under addition, i.e. any $k_1\ Q\ k_2$ and $k_1'\ Q\ k_2'$ satisfy $(k_1 + k_1')\ Q\ (k_2 + k_2')$.

(B) $Q' \subseteq \geq_{\mathbb{N}}$, and $Q' \subseteq Q''$.

(C) $Q' \circ Q \circ Q'' \subseteq Q$, where $\circ$ denotes composition of binary relations.

Examples of a reasonable triple $(Q, Q', Q'')$ include: $(\mathbb{N} \times \mathbb{N}, \geq_{\mathbb{N}}, \mathbb{N} \times \mathbb{N})$, $(\mathbb{N} \times \mathbb{N}, \geq_{\mathbb{N}}, \geq_{\mathbb{N}})$, $(\mathbb{N} \times \mathbb{N}, =_{\mathbb{N}}, =_{\mathbb{N}})$, $(\geq_{\mathbb{N}}, \geq_{\mathbb{N}}, \geq_{\mathbb{N}})$, $(\geq_{\mathbb{N}}, =_{\mathbb{N}}, \geq_{\mathbb{N}})$, $(\leq_{\mathbb{N}}, =_{\mathbb{N}}, \leq_{\mathbb{N}})$, $(\geq_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$, $(\leq_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$, $(=_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$.

**Theorem 4.3.14.** *If a universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ is deterministic and refocusing, it satisfies the following property. For any set $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$ of contexts that is closed under plugging, any reasonable triple $(Q, Q', Q'')$, and any pre-template $\lhd$ on focus-free hypernets $\mathcal{H}_\omega(L, M_{\mathbb{O}} \backslash \{?, \checkmark, \lightning\})$:*

1. *If $\lhd$ is a $(\mathbb{C}, Q, Q')$-template and $(\mathbb{C}, Q, Q', Q'')$-robust relative to all rewrite transitions, then $\lhd$ implies contextual refinement in $\mathbb{C}$ up to $Q$, i.e. any $G_1 \lhd G_2$ implies $G_1 \preceq_Q^{\mathbb{C}} G_2$.*

2. *If $\lhd$ is a $(\mathbb{C}, Q^{-1}, Q')$-template and the converse $\lhd^{-1}$ is $(\mathbb{C}, Q, Q', Q'')$-robust relative to all rewrite transitions, then $\lhd^{-1}$ implies contextual refinement in $\mathbb{C}$ up to $Q$, i.e. any $G_1 \lhd G_2$ implies $G_2 \preceq_Q^{\mathbb{C}} G_1$.*

*Proof.* This is a consequence of Prop. 4.4.6, Prop. 4.4.2 and Prop. 4.4.4 in Sec. 4.4. □

**Remark 4.3.15** (Monotonicity)**.** *Contextual/state refinement and equivalence are monotonic with respect to $Q$, in the sense that $Q_1 \subseteq Q_2$ implies $\square_{Q_1} \subseteq \square_{Q_2}$ for each $\square \in \{\dot{\preceq}, \dot{\simeq}, \preceq^{\mathbb{C}}, \simeq^{\mathbb{C}}\}$. Contextual refinement and equivalence are anti-monotonic with respect to $\mathbb{C}$, in the sense that $\mathbb{C}_1 \subseteq \mathbb{C}_2$ implies $\square^{\mathbb{C}_2} \subseteq \square^{\mathbb{C}_1}$ for each $\square \in \{\preceq_Q, \simeq_Q\}$. This means, in particular, $\simeq_Q^{\mathbb{C}_{\mathbb{O}}} \subseteq \simeq_Q^{\mathbb{C}_{\mathbb{O}\text{-bf}}}$.*

*Given that $\mathbb{C}_1 \subseteq \mathbb{C}_2$, $Q_1 \subseteq Q_2$, $Q_1' \subseteq Q_2'$, $Q_1' \subseteq Q_2'$, $R_1 \subseteq R_2$ and $R_1' \subseteq R_2'$, the following holds. Any $\mathbb{C}_1$-specimen is a $\mathbb{C}_2$-specimen, and any quasi-$\mathbb{C}_1$-specimen up to $(R_1, R_1')$ is a quasi-$\mathbb{C}_2$-specimen up to $(R_2, R_2')$. Any $(\mathbb{C}, Q_1, Q_1')$-template is a $(\mathbb{C}, Q_2, Q_2')$-template. If $\lhd$ is $(\mathbb{C}, Q_1, Q_1', Q_1'')$-robust relative to a rewrite transition, then it is also $(\mathbb{C}, Q_2, Q_2', Q_2'')$-robust relative to the same transition. Note that the notions of template and robustness are not monotonic nor anti-monotonic with respect to $\mathbb{C}$.*

*To prove that a pre-template $\lhd$ induces contextual equivalence, one can use Thm. 4.3.14(1) twice with respect to $\lhd$ and $\lhd^{-1}$. One can alternatively use Thm. 4.3.14(1) and Thm. 4.3.14(2), both with respect to $\lhd$. This alternative approach is often more economical. The reason is that the approach involves proving input-safety of $\lhd$ with respect to two parameters $(\mathbb{C}, Q, Q')$ and $(\mathbb{C}, Q^{-1}, Q')$, which typically boils down to a proof for one parameter, thanks to the monotonicity.*

### 4.3.3 Sufficient conditions for robustness

A proof of robustness becomes trivial for a specimen with a rewrite token that gives a non-rooted state. Thanks to the lemma below, we can show that a state is not rooted, by checking paths from the token target.

**Definition 4.3.16** (Accessible paths)**.**

- A path of a hypernet is said to be *accessible* if it consists of edges whose all sources have type $\star$.
- An accessible path is called *stable* if the labels of its edges are included in $\{\mathsf{I}\} \cup \mathbb{O}_{\checkmark}$.
- An accessible path is called *active* if it starts with one active operation edge and possibly followed by a stable path.

Note that box edges and atom edges never appear in an accessible path.

**Lemma 4.3.17.** *If a state has a rewrite token that is not an incoming edge of a contraction edge, then the state satisfies the following property: If there exists an accessible, but not active, path from the token target, then the state is not rooted.*

*Proof.* This is a contraposition of a consequence of Lem. A.3.5 and Lem. A.4.6(2). □

Checking the condition (III) of robustness (see Def. 4.3.11) involves finding a quasi-$\mathbb{C}$-specimen of $\lhd$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, namely checking the condition (B) of Def. 4.3.6(2). The following lemma enables us to use contextual refinement $\preceq^{\mathbb{C}}_{Q'}$ to yield state refinement $\dot{\preceq}_{Q'}$, via single $\mathbb{C}$-specimens of a certain pre-template $\lhd$.

**Definition 4.3.18.** A pre-template $\lhd$ is a *trigger* if it satisfies the following:

(A) For any single $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\chi]; H^1; H^2)$ of $\lhd$, such that $\dot{\mathcal{C}}$ has an entering search token, $\dot{\mathcal{C}}[H^p] \to \langle \dot{\mathcal{C}}[H^p] \rangle_{\lightning/?}$ for each $p \in \{1, 2\}$.

(B) For any hypernets $H^1 \lhd H^2$, both $H_1$ and $H_2$ are one-way.

**Lemma 4.3.19.** *Let $\mathbb{C}$ be a set of contexts, and $Q'$ be a binary relation on $\mathbb{N}$ such that, for any $k_0, k_1, k_2 \in \mathbb{N}$, $(k_0 + k_1)\ Q'\ (k_0 + k_2)$ implies $k_1\ Q'\ k_2$. Let $\lhd$ be a pre-template that is a trigger and implies contextual refinement $\preceq^{\mathbb{C}}_{Q'}$. For any single $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\chi]; H^1; H^2)$ of $\lhd$, if compute transitions are all deterministic, and one of states $\dot{\mathcal{C}}[H^1]$ and $\dot{\mathcal{C}}[H^2]$ is rooted, then the other state is also rooted, and moreover, $\dot{\mathcal{C}}[H^1] \dot{\preceq}_{Q'} \dot{\mathcal{C}}[H^2]$.*

*Proof.* This is a corollary of Lem. A.6.1. □

**Remark 4.3.20.** *The notion of contextual refinement concerns initial states, and therefore, only enables us to safely replace a part of a hypernet* before *execution. Because any initial state is rooted, if all transitions preserve the rooted property, we can safely assume that any state that arises in an execution is rooted. If all transitions are also deterministic, Lem. 4.3.19 enables us to use some contextual refinement and safely replace a part of a hypernet* during *execution. This can validate run-time garbage collection, for example.*

## 4.4 Proof of the characterisation theorem

This section details the proof of Thm. 4.3.14, with respect to the machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ parametrised by $\mathbb{O}$ and $B_{\mathbb{O}}$.

At the core of the proof is step-wise reasoning, or transition-wise reasoning, using a lax variation of simulation. Providing a simulation boils down to case analysis on transitions, namely on possible interactions between the token and parts of states contributed by a pre-template. While output-closure helps us disprove some cases under the assumption that the machine is refocusing, input-safety and robustness deal with the cases that are specific to a pre-template and an operation set.

The step-wise reasoning is enriched with the so-called *up-to* technique (see e.g. Milner [1999]). Our variation of simulation is namely up to state refinements, which is reflected in the definition of quasi-specimen (Def. 4.3.6(2)). In order to make this particular up-to technique work, it is essential to additionally equip our simulation with a quantitative restriction. The restriction is implemented by the notion of reasonable triple. A similar form of up-to technique is studied categorically by Bonchi et al. [2017], but for the ordinary notion of (weak) simulation, without this quantitative restriction.

The lax variation of simulation we use is namely $(Q, Q', Q'')$-*simulation*, parametrised by a triple $(Q, Q', Q'')$. This provides a sound approach to prove state refinement $\dot{\preceq}_Q$, using $\dot{\preceq}_{Q'}$ and $\dot{\preceq}_{Q''}$, given that all transitions are deterministic and $(Q, Q', Q'')$ forms a reasonable triple.

**Definition 4.4.1** ($(Q, Q', Q'')$-simulations)**.** Let $R$ be a binary relation on states, and $(Q, Q', Q'')$ be a triple of preorders on $\mathbb{N}$. The binary relation $R$ is a $(Q, Q', Q'')$-*simulation* if, for any two related states $\dot{G}_1\ R\ \dot{G}_2$, the following (A) and (B) hold:

(A) If $\dot{G}_1$ is final, $\dot{G}_2$ is also final.

(B) If there exists a state $\dot{G}_1'$ such that $\dot{G}_1 \to \dot{G}_1'$, one of the following (I) and (II) holds:

(I) There exists a stuck state $\dot{G}_1''$ such that $\dot{G}_1' \to^* \dot{G}_1''$.

(II) There exist two states $\dot{H}_1$ and $\dot{H}_2$, and numbers $k_1, k_2 \in \mathbb{N}$, such that $\dot{H}_1\ (\dot{\preceq}_{Q'} \circ R \circ \dot{\preceq}_{Q''})\ \dot{H}_2$, $(1 + k_1)\ Q\ k_2$, $\dot{G}_1' \to^{k_1} \dot{H}_1$, and $\dot{G}_2 \to^{k_2} \dot{H}_2$.

**Proposition 4.4.2.** *When the universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ is deterministic, it satisfies the following.*

*For any binary relation $R$ on states, and any reasonable triple $(Q, Q', Q'')$, if $R$ is a $(Q, Q', Q'')$-simulation, then $R$ implies refinement up to $Q$, i.e. any $\dot{G}_1\ R\ \dot{G}_2$ implies $\dot{G}_1 \mathrel{\dot{\preceq}_Q} \dot{G}_2$.*

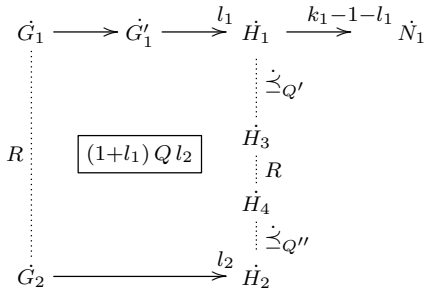
*Proof.* Our goal is to show the following: for any states $\dot{G}_1\ R\ \dot{G}_2$, any number $k_1 \in \mathbb{N}$ and any final state $\dot{N}_1$, such that $\dot{G}_1 \to^{k_1} \dot{N}_1$, there exist a number $k_2 \in \mathbb{N}$ and a final state $\dot{N}_2$ such that $k_1\ Q\ k_2$ and $\dot{G}_2 \to^{k_2} \dot{N}_2$. The proof is by induction on $k_1 \in \mathbb{N}$.

In the base case, when $k_1 = 0$, the state $\dot{G}_1$ is itself final because $\dot{G}_1 = \dot{N}_1$. Because $R$ is a $(Q, Q', Q'')$-simulation, $\dot{G}_2$ is also a final state, which means we can take 0 as $k_2$ and $\dot{G}_2$ itself as $\dot{N}_2$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q$ is a preorder and $0\ Q\ 0$ holds.

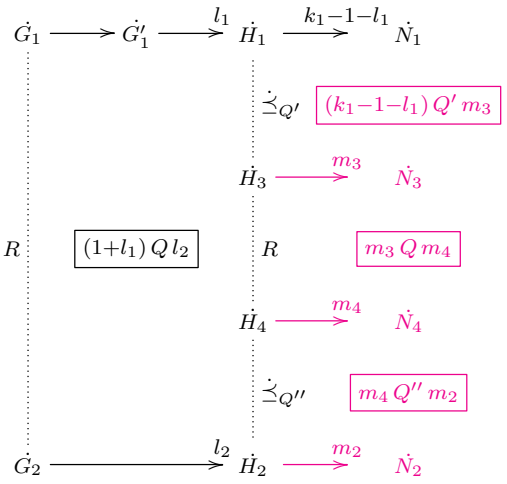
In the inductive case, when $k_1 > 0$, we assume the induction hypothesis on any $h \in \mathbb{N}$ such that $h < k_1$. Now that $k_1 > 0$, there exists a state $\dot{G}'_1$ such that $\dot{G}_1 \to \dot{G}'_1 \to^{k-1} \dot{N}_1$. Because all intrinsic transitions are deterministic, the assumption that compute transitions are all deterministic implies that states and transitions comprise a deterministic abstract rewriting system, in which final states and stuck states are normal forms. By Lem. A.3.1, we can conclude that there exists no stuck state $\dot{G}''_1$ such that $\dot{G}'_1 \to^* \dot{G}''_1$.

Therefore, by $R$ being a $(Q, Q', Q'')$-simulation, there exist two states $\dot{H}_1$ and $\dot{H}_2$, and numbers $l_1, l_2 \in \mathbb{N}$, such that $\dot{H}_1\ (\mathrel{\dot{\preceq}_{Q'}} \circ R \circ \mathrel{\dot{\preceq}_{Q''}})\ \dot{H}_2$, $(1 + l_1)\ Q\ l_2$, $\dot{G}'_1 \to^{l_1} \dot{H}_1$, and $\dot{G}_2 \to^{l_2} \dot{H}_2$. By the determinism, $1 + l_1 \leq k_1$ must hold; if $\dot{H}_1$ is a final state, $\dot{G}'_1 \to^{l_1} \dot{H}_1$ must coincide with $\dot{G}'_1 \to^{k-1} \dot{N}_1$; otherwise, $\dot{G}'_1 \to^{l_1} \dot{H}_1$ must be a suffix of $\dot{G}'_1 \to^{k-1} \dot{N}_1$. There exist two states $\dot{H}_3$ and $\dot{H}_4$, and we have the following situation, where the relations $R$, $\mathrel{\dot{\preceq}_{Q'}}$ and $\mathrel{\dot{\preceq}_{Q''}}$ are represented by vertical dotted

lines from top to bottom.




We expand the above diagram as below (indicated by magenta), in three steps.




Firstly, by definition of state refinement, there exist a number $m_3 \in \mathbb{N}$ and a final state $\dot{N}_3$ such that $(k_1 - 1 - l_1)\ Q'\ m_3$ and $\dot{H}_3 \rightarrow^{m_3} \dot{N}_3$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q' \subseteq \geq_{\mathbb{N}}$, and hence $k_1 > k_1 - 1 - l_1 \geq m_3$. Therefore, secondly, by induction hypothesis on $m_3$, there exist a number $m_4 \in \mathbb{N}$ and a final state $\dot{N}_4$ such that $m_3\ Q\ m_4$ and $\dot{H}_4 \rightarrow^{m_4} \dot{N}_4$. Thirdly, by definition of state refinement, there exist a number $m_2 \in \mathbb{N}$ and a final state $\dot{N}_2$ such that $m_4\ Q''\ m_2$ and $\dot{H}_2 \rightarrow^{m_2} \dot{N}_2$.

Now we have $(k_1 - 1 - l_1)\ Q'\ m_3$, $m_3\ Q\ m_4$ and $m_4\ Q''\ m_2$, which means $(k_1 - 1 - l_1)\ (Q' \circ Q \circ Q'')\ m_2$. Because $(Q, Q', Q'')$ is a reasonable triple, this implies $(k_1 - 1 - l_1)\ Q\ m_2$, and moreover, $k_1\ Q\ (l_2 + m_2)$. We can take $l_2 + m_2$ as $k_2$. □

The token in a focussed context $\dot{\mathcal{C}}$ is said to be *remote*, if it is a search token, a value token, or not entering. The procedure of *contextual lifting* reduces a proof of contextual refinement down to that of state refinement.

**Definition 4.4.3** (Contextual lifting)**.** Let $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_\mathbb{O} \cup \mathbb{M})$ be a set of contexts. Given a pre-template $\lhd$ on focus-free hypernets $\mathcal{H}_\omega(L, M_\mathbb{O} \backslash \{?, \checkmark, \lightning\})$, its $\mathbb{C}$-*contextual lifting* $\overline{\lhd}^{\mathbb{C}}$ is a binary relation on states defined by: $\dot{G}_1 \,\overline{\lhd}^{\mathbb{C}}\, \dot{G}_2$ if there exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ of $\lhd$, such that the token of $\dot{\mathcal{C}}$ is remote, $\dot{G}_p = \dot{\mathcal{C}}[\vec{H}^p]$, and $\dot{\mathcal{C}}[\vec{H}^p]$ is rooted, for each $p \in \{1, 2\}$.

The contextual lifting $\overline{\lhd}^{\mathbb{C}}$ is by definition a binary relation on rooted states.

**Proposition 4.4.4.** *For any set $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_\mathbb{O} \cup \mathbb{M})$ of contexts that is closed under plugging, any preorder $Q$ on $\mathbb{N}$, and any pre-template $\lhd$ on focus-free hypernets $\mathcal{H}_\omega(L, M_\mathbb{O} \backslash \{?, \checkmark, \lightning\})$, if the $\mathbb{C}$-contextual lifting $\overline{\lhd}^{\mathbb{C}}$ implies refinement $\dot{\preceq}_Q$ (resp. equivalence $\dot{\simeq}_Q$), then $\lhd$ implies contextual refinement $\preceq_Q^{\mathbb{C}}$ (resp. contextual equivalence $\simeq_Q^{\mathbb{C}}$).*

*Proof of refinement case.* Our goal is to show that, for any $H_1 \lhd H_2$ and any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$ such that $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are states, we have refinement $?;\mathcal{C}[H_1] \,\dot{\preceq}_Q\, ?;\mathcal{C}[H_2]$.

Because $?;\mathcal{C}[H_p] = ?;(\mathcal{C}[H_p]) = (?;\mathcal{C})[H_p]$ for $p \in \{1, 2\}$, and $|?;\mathcal{C}| = \mathcal{C} \in \mathbb{C}$, the triple $((?;\mathcal{C}); H_1; H_2)$ is a $\mathbb{C}$-specimen of $\lhd$ with a search token. Moreover the states $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are trivially rooted. Therefore, $?;\mathcal{C}[H_1] \,\overline{\lhd}^{\mathbb{C}}\, ?;\mathcal{C}[H_2]$, and by the assumption, $?;\mathcal{C}[H_1] \,\dot{\preceq}_Q\, ?;\mathcal{C}[H_2]$. □

*Proof of equivalence case.* It suffices to show that, for any $H_1 \lhd H_2$ and any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$ such that $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are states, we have refinements $?;\mathcal{C}[H_1] \,\dot{\preceq}_Q\, ?;\mathcal{C}[H_2]$ and $?;\mathcal{C}[H_2] \,\dot{\preceq}_Q\, ?;\mathcal{C}[H_1]$, i.e. equivalence $?;\mathcal{C}[H_1] \,\dot{\simeq}_Q\, ?;\mathcal{C}[H_2]$.

Because $?;\mathcal{C}[H_p] = ?;(\mathcal{C}[H_p]) = (?;\mathcal{C})[H_p]$ for $p \in \{1, 2\}$, and $|?;\mathcal{C}| = \mathcal{C} \in \mathbb{C}$, the triple $((?;\mathcal{C}); H_1; H_2)$ is a $\mathbb{C}$-specimen of $\lhd$ with a search token. Moreover the states

$?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are trivially rooted. Therefore, $?;\mathcal{C}[H_1] \mathrel{\overline{\lhd}^{\mathbb{C}}} ?;\mathcal{C}[H_2]$, and by the assumption, $?;\mathcal{C}[H_1] \mathrel{\dot{\simeq}_Q} ?;\mathcal{C}[H_2]$. □

**Lemma 4.4.5.** *For any set $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$ of contexts that is closed under plugging, any pre-template $\lhd$ on focus-free hypernets $\mathcal{H}_\omega(L, M_{\mathbb{O}} \backslash \{?, \checkmark, \lightning\})$, and any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\vec{\chi}]; \vec{H}^1; \vec{H}^2)$ of $\lhd$, the following holds.*

1. *The state $\dot{\mathcal{C}}[\vec{H}^1]$ is final (resp. initial) if and only if the state $\dot{\mathcal{C}}[\vec{H}^2]$ is final (resp. initial).*

2. *If $\lhd$ is output-closed, and $\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$ are both rooted states, then the token of $\dot{\mathcal{C}}$ is not exiting.*

3. *If $\lhd$ is output-closed, $\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$ are both rooted states, the token of $\dot{\mathcal{C}}$ is a value token or a non-entering search token, and a transition is possible from $\dot{\mathcal{C}}[\vec{H}^1]$ or $\dot{\mathcal{C}}[\vec{H}^2]$, then there exists a focussed context $\dot{\mathcal{C}}'$ with a remote token such that $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$ and $\dot{\mathcal{C}}[\vec{H}^p] \to \dot{\mathcal{C}}'[\vec{H}^p]$ for each $p \in \{1, 2\}$.*

*Proof of point (1).* Let $(p, q)$ be an arbitrary element of a set $\{(1, 2), (2, 1)\}$. If $\dot{\mathcal{C}}[\vec{H}^p]$ is final (resp. initial), the token source is an input in $\dot{\mathcal{C}}[\vec{H}^p]$. Because input lists of $\dot{\mathcal{C}}[\vec{H}^p]$, $\dot{\mathcal{C}}$ and $\dot{\mathcal{C}}[\vec{H}^q]$ all coincide, the token source must be an input in $\dot{\mathcal{C}}$, and in $\dot{\mathcal{C}}[\vec{H}^q]$ too. This means $\dot{\mathcal{C}}[\vec{H}^q]$ is also a final (resp. initial) state. □

*Proof of point (2).* This is a consequence of the contraposition of Lem. A.3.7(3). □

*Proof of the point (3).* The transition possible from $\dot{\mathcal{C}}[\vec{H}^1]$ or $\dot{\mathcal{C}}[\vec{H}^2]$ is necessarily a search transition. By case analysis on the token of $\dot{\mathcal{C}}$, we can confirm that the search transition applies an interaction rule to the token and an edge from $\dot{\mathcal{C}}$.

- When the token of $\dot{\mathcal{C}}$ is a value token, the transition can only change the token and its incoming operation edge. Because $\lhd$ is output-closed, by the point (2), the token of $\dot{\mathcal{C}}$ is not exiting. This implies that the incoming operation edge of the token is from $\dot{\mathcal{C}}$ in both states $\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$.

- When the token of $\dot{\mathcal{C}}$ is a non-entering search token, the transition can only change the token and its outgoing edge. Because the token is not entering in $\dot{\mathcal{C}}$, the outgoing edge is from $\dot{\mathcal{C}}$ in both states $\dot{\mathcal{C}}[\vec{H}^1]$ and $\dot{\mathcal{C}}[\vec{H}^2]$.

Therefore, there exist a focus-free simple context $\mathcal{C}_0[\chi, \vec{\chi}]$ and an interaction rule $\dot{N}_0 \mapsto \dot{N}_0'$, such that $\dot{\mathcal{C}} = \mathcal{C}_0[\dot{N}_0, \vec{\chi}]$, and $\mathcal{C}_0[\dot{N}_0', \vec{\chi}]$ is a focussed context.

Examining interaction rules confirms $|\dot{N}_0| = |\dot{N}_0'|$, and hence $|\dot{\mathcal{C}}| = |\mathcal{C}_0[\dot{N}_0, \vec{\chi}]| = |\mathcal{C}_0[\dot{N}_0', \vec{\chi}]|$. By definition of search transitions, we have:

$$\dot{\mathcal{C}}[\vec{H}^p] = \mathcal{C}_0[\dot{N}_0, \vec{H}^p] \to \mathcal{C}_0[\dot{N}_0', \vec{H}^p]$$

for each $p \in \{1, 2\}$.

The rest of the proof is to check that $\mathcal{C}_0[\dot{N}_0', \vec{\chi}]$ has a remote token, namely that, if its token is a rewrite token, the token is not entering. This is done by inspecting interaction rules.

- When the interaction rule $\dot{N}_0 \mapsto \dot{N}_0'$ changes a value token to a rewrite token, this must be the interaction rule (5a), which means $\dot{N}_0'$ consists of the rewrite token and its outgoing operation edge. The operation edge remains to be a (unique) outgoing edge of the token in $\mathcal{C}_0[\dot{N}_0', \vec{\chi}]$, and hence the token is not entering in $\mathcal{C}_0[\dot{N}_0', \vec{\chi}]$.

- When the interaction rule $\dot{N}_0 \mapsto \dot{N}_0'$ changes a search token to a rewrite token, this must be the interaction rule (1a), (1b) or (5b), which means $\dot{N}_0' = \langle \dot{N}_0 \rangle_{\lightning/?}$. Because the token is not entering in $\mathcal{C}_0[\dot{N}_0, \vec{\chi}] = \dot{\mathcal{C}}$, the token is also not entering in $\mathcal{C}_0[\dot{N}_0', \vec{\chi}] = \langle \mathcal{C}_0[\dot{N}_0, \vec{\chi}] \rangle_{\lightning/?}$.

□

**Proposition 4.4.6.** *When the universal abstract machine $\mathcal{U}(\mathbb{O}, B_{\mathbb{O}})$ is deterministic and refocusing, it satisfies the following, for any set $\mathbb{C} \subseteq \mathcal{H}_\omega(L, M_{\mathbb{O}} \cup \mathbb{M})$ of contexts*

*that is closed under plugging, any reasonable triple* $(Q, Q', Q'')$*, and any pre-template* $\lhd$ *on focus-free hypernets* $\mathcal{H}_\omega(L, M_{\mathbb{O}} \backslash \{?, \checkmark, \lightning\})$.

1. *If* $\lhd$ *is a* $(\mathbb{C}, Q, Q')$*-template and* $(\mathbb{C}, Q, Q', Q'')$*-robust relative to all rewrite transitions, then the* $\mathbb{C}$*-contextual lifting* $\overline{\lhd}^{\mathbb{C}}$ *is a* $(Q, Q', Q'')$*-simulation.*

2. *If* $\lhd$ *is a* $(\mathbb{C}, Q^{-1}, Q')$*-template and the converse* $\lhd^{-1}$ *is* $(\mathbb{C}, Q, Q', Q'')$*-robust relative to all rewrite transitions, then the* $\mathbb{C}$*-contextual lifting* $\overline{\lhd^{-1}}^{\mathbb{C}}$ *of the converse is a* $(Q, Q', Q'')$*-simulation.*

*Proof prelude.* Let $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ be an arbitrary $\mathbb{C}$-specimen of $\lhd$, such that the token of $\dot{\mathcal{C}}$ is remote, and $\dot{G}_p := \dot{\mathcal{C}}[\vec{H}^p]$ is a rooted state for each $p \in \{1, 2\}$. By definition of contextual lifting, $\dot{G}_1 \; \overline{\lhd}^{\mathbb{C}} \; \dot{G}_2$, and equivalently, $\dot{G}_2 \; (\overline{\lhd}^{\mathbb{C}})^{-1} \; \dot{G}_1$. Note that $\overline{\lhd^{-1}}^{\mathbb{C}} = (\overline{\lhd}^{\mathbb{C}})^{-1}$.

Because $\lhd$ is output-closed, by Lem. 4.4.5(2), the token is not exiting in $\dot{\mathcal{C}}$. This implies that, if the token has an incoming edge in $\dot{G}_1$ or $\dot{G}_2$, the incoming edge must be from $\dot{\mathcal{C}}$.

Because the machine is deterministic and refocusing, rooted states and transitions comprise a deterministic abstract rewriting system, in which final states and stuck states are normal forms. By Lem. A.3.1, from any state, a sequence of transitions that results in a final state or a stuck state is unique, if any.

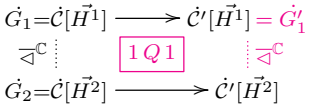
Because $(Q, Q', Q'')$ is a reasonable triple, $Q'$ and $Q''$ are reflexive. By Lem. A.5.2, this implies that $\dot{\preceq}_{Q'}$ and $\dot{\preceq}_{Q''}$ are reflexive, and hence $\overline{\lhd}^{\mathbb{C}} \subseteq \dot{\preceq}_{Q'} \circ \overline{\lhd}^{\mathbb{C}} \circ \dot{\preceq}_{Q''}$, and $(\overline{\lhd}^{\mathbb{C}})^{-1} \subseteq \dot{\preceq}_{Q'} \circ (\overline{\lhd}^{\mathbb{C}})^{-1} \circ \dot{\preceq}_{Q''}$. □

*Proof of the point (1).* Our goal is to check conditions (A) and (B) of Def. 4.4.1 for the states $\dot{G}_1 \; \overline{\lhd}^{\mathbb{C}} \; \dot{G}_2$.

If $\dot{G}_1$ is final, by Lem. 4.4.5(1), $\dot{G}_2$ is also final. The condition (A) of Def. 4.4.1 is fulfilled.

If there exists a state $\dot{G}'_1$ such that $\dot{G}_1 \to \dot{G}'_1$, we show that one of the conditions (I) and (II) of Def. 4.4.1 is fulfilled, by case analysis of the token in $\dot{\mathcal{C}}$.

- When the token is a value token, or a search token that is not entering, by Lem. 4.4.5(3), there exists a focussed context $\dot{\mathcal{C}}'$ with a remote token, such that $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$ and $\dot{G}_p = \dot{\mathcal{C}}[\vec{H}^p] \to \dot{\mathcal{C}}'[\vec{H}^p]$ for each $p \in \{1, 2\}$. We have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II) of Def. 4.4.1 is fulfilled.

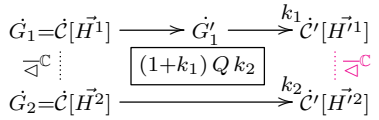



  By the determinism, $\dot{\mathcal{C}}'[\vec{H}^1] = \dot{G}'_1$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q$ is a preorder and $1\ Q\ 1$. The context $\dot{\mathcal{C}}'$ satisfies $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}| \in \mathbb{C}$, so $(\dot{\mathcal{C}}'; \vec{H}^1; \vec{H}^2)$ is a $\mathbb{C}$-specimen of $\lhd$. The context $\dot{\mathcal{C}}'$ has a remote token, and the states $\dot{\mathcal{C}}'[\vec{H}^1]$ and $\dot{\mathcal{C}}'[\vec{H}^2]$ are both rooted. Therefore, we have $\dot{\mathcal{C}}'[\vec{H}^1] \trianglelefteq^{\mathbb{C}} \dot{\mathcal{C}}'[\vec{H}^2]$.

- When the token is a search token that is entering in $\dot{\mathcal{C}}$, because $\lhd$ is $(\mathbb{C}, Q, Q')$-input-safe, we have one of the following three situations corresponding to (I), (II) and (III) of Def. 4.3.7.

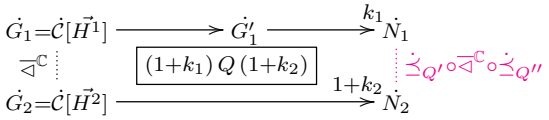
  - There exist two stuck states $\dot{N}_1$ and $\dot{N}_2$ such that $\dot{G}_p \to^* \dot{N}_p$ for each $p \in \{1, 2\}$. By the determinism of transitions, we have $\dot{G}_1 \to \dot{G}'_1 \to^* \dot{N}_1$, which means the condition (I) of Def. 4.4.1 is satisfied.

  - There exist a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ and two numbers $k_1, k_2 \in \mathbb{N}$, such that the token of $\dot{\mathcal{C}}'$ is not a rewrite token and not entering, $(1{+}k_1)\ Q\ k_2$, $\dot{\mathcal{C}}[\vec{H}^1] \to^{1+k_1} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}[\vec{H}^2] \to^{k_2} \dot{\mathcal{C}}'[\vec{H'}^2]$. By the determinism of transitions, we have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II) of Def. 4.4.1 is fulfilled.



The context $\dot{\mathcal{C}}'$ has a remote token, and states $\dot{\mathcal{C}}'[\vec{H'}^1]$ and $\dot{\mathcal{C}}'[\vec{H'}^2]$ are rooted. Therefore, $\dot{\mathcal{C}}'[\vec{H'}^1] \mathrel{\overline{\lhd}^{\mathbb{C}}} \dot{\mathcal{C}}'[\vec{H'}^2]$.

– There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_1, \dot{N}_2)$ of $\lhd$ up to $(\dot{\simeq}_{Q'}, \dot{\simeq}_{Q'})$, whose token is not a rewrite token, and two numbers $k_1, k_2 \in \mathbb{N}$, such that $(1 + k_1)\ Q\ (1 + k_2)$, $\dot{\mathcal{C}}[\vec{H}^1] \rightarrow^{1+k_1} \dot{N}_1$, and $\dot{\mathcal{C}}[\vec{H}^2] \rightarrow^{1+k_2} \dot{N}_2$. By the determinism of transitions, we have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II) of Def. 4.4.1 is fulfilled.




Because $(\dot{N}_1, \dot{N}_2)$ is a quasi-$\mathbb{C}$-specimen of $\lhd$ up to $(\dot{\simeq}_{Q'}, \dot{\simeq}_{Q'})$, and states $\dot{N}_1$ and $\dot{N}_2$ are rooted, there exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ with a non-rewrite token, such that $\dot{\mathcal{C}}'[\vec{H'}^1]$ and $\dot{\mathcal{C}}'[\vec{H'}^2]$ are also rooted, $\dot{N}_1 \dot{\simeq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}'[\vec{H'}^2] \dot{\simeq}_{Q'} \dot{N}_2$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q' \subseteq Q''$, and hence $\dot{\simeq}_{Q'} \subseteq \dot{\simeq}_{Q''}$. Therefore, we have:

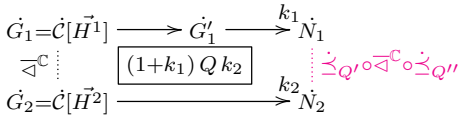
$$\dot{N}_1 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^1] \mathrel{\overline{\lhd}^{\mathbb{C}}} \dot{\mathcal{C}}'[\vec{H'}^2] \dot{\preceq}_{Q''} \dot{N}_2.$$

- When the token is a rewrite token, $\dot{G}_1 \rightarrow \dot{G}'_1$ is a rewrite transition, and by definition of contextual lifting, the token is not entering in $\dot{\mathcal{C}}$. Because $\lhd$ is $(\mathbb{C}, Q, Q', Q'')$-robust relative to all rewrite transitions, and $\dot{G}_1$ and $\dot{G}_2$ are rooted, we have one of the following two situations corresponding to (II) and (III) of Def. 4.3.11.

  – There exists a stuck state $\dot{N}$ such that $\dot{G}'_1 \rightarrow^* \dot{N}$. The condition (I) of Def. 4.4.1 is satisfied.

  – There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_1, \dot{N}_2)$ of $\lhd$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, whose

token is not a rewrite token, and two numbers $k_1, k_2 \in \mathbb{N}$, such that $(1 + k_1)\ Q\ k_2$, $\dot{G}_1' \to^{k_1} \dot{N}_1$, and $\dot{G}_2 \to^{k_2} \dot{N}_2$. We have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II) of Def. 4.4.1 is fulfilled.




Because $(\dot{N}_1, \dot{N}_2)$ is a quasi-$\mathbb{C}$-specimen of $\lhd$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, and states $\dot{N}_1$ and $\dot{N}_2$ are rooted, there exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ with a non-rewrite token, such that $\dot{\mathcal{C}}'[\vec{H'}^1]$ and $\dot{\mathcal{C}}'[\vec{H'}^2]$ are also rooted, $\dot{N}_1 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}'[\vec{H'}^2] \dot{\preceq}_{Q''} \dot{N}_2$. This means $\dot{\mathcal{C}}'[\vec{H'}^1] \overline{\lhd}^{\mathbb{C}} \dot{\mathcal{C}}'[\vec{H'}^2]$, and hence:

$$\dot{N}_1 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^1] \overline{\lhd}^{\mathbb{C}} \dot{\mathcal{C}}'[\vec{H'}^2] \dot{\preceq}_{Q''} \dot{N}_2.$$

□

*Proof of the point (2).* It suffices to check the reverse of conditions (A) and (B) of Def. 4.4.1 for the states $\dot{G}_2\ (\overline{\lhd}^{\mathbb{C}})^{-1}\ \dot{G}_1$, namely the following conditions (A') and (B').

(A') If $\dot{G}_2$ is final, $\dot{G}_1$ is also final.

(B') If there exists a state $\dot{G}_2'$ such that $\dot{G}_2 \to \dot{G}_2'$, one of the following (I') and (II') holds.

(I') There exists a stuck state $\dot{G}_2''$ such that $\dot{G}_2' \to^* \dot{G}_2''$.

(II') There exist two states $\dot{N}_2$ and $\dot{N}_1$, and numbers $k_2, k_1 \in \mathbb{N}$, such that $\dot{N}_2\ (\dot{\preceq}_{Q'} \circ (\overline{\lhd}^{\mathbb{C}})^{-1} \circ \dot{\preceq}_{Q'})\ \dot{N}_1$, $(1 + k_2)\ Q\ k_1$, $\dot{G}_2' \to^{k_2} \dot{N}_2$, and $\dot{G}_1 \to^{k_1} \dot{N}_1$.

The proof is mostly symmetric to the point (1). Note that there is a one-to-one correspondence between $\mathbb{C}$-specimens of $\lhd$ and $\mathbb{C}$-specimens of $\lhd^{-1}$; any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}_0; \vec{H}^{01}; \vec{H}^{02})$ of $\lhd$ gives a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}_0; \vec{H}^{02}; \vec{H}^{01})$ of $\lhd^{-1}$. Because $\lhd$ is output-closed, its converse $\lhd^{-1}$ is also output-closed.

If $\dot{G}_2$ is final, by Lem. 4.4.5(1), $\dot{G}_1$ is also final. The condition (A') is fulfilled.

If there exists a state $\dot{G}'_2$ such that $\dot{G}_2 \to \dot{G}'_2$, we show that one of the conditions (I') and (II') above is fulfilled, by case analysis of the token in $\dot{\mathcal{C}}$.

- When the token is a value token, or a search token that is not entering, by Lem. 4.4.5(3), there exists a focussed context $\dot{\mathcal{C}}'$ with a remote token, such that $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$ and $\dot{G}_p = \dot{\mathcal{C}}[\vec{H}^p] \to \dot{\mathcal{C}}'[\vec{H}^p]$ for each $p \in \{1, 2\}$. We have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II') is fulfilled.

$$\begin{array}{ccc} \dot{G}_2{=}\dot{\mathcal{C}}[\vec{H}^1] & \longrightarrow & \dot{\mathcal{C}}'[\vec{H}^2] = \dot{G}'_2 \\ (\lhd^{\mathbb{C}})^{-1} \vdots & \boxed{1\,Q\,1} & \vdots (\lhd^{\mathbb{C}})^{-1} \\ \dot{G}_1{=}\dot{\mathcal{C}}[\vec{H}^1] & \longrightarrow & \dot{\mathcal{C}}'[\vec{H}^1] \end{array}$$

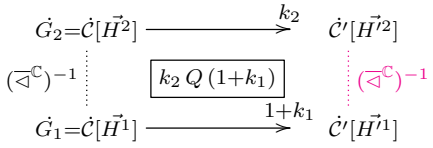
  By the determinism, $\dot{\mathcal{C}}'[\vec{H}^2] = \dot{G}'_2$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q$ is a preorder and $1\ Q\ 1$. The context $\dot{\mathcal{C}}'$ satisfies $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}| \in \mathbb{C}$, so $(\dot{\mathcal{C}}'; \vec{H}^2; \vec{H}^1)$ is a $\mathbb{C}$-specimen of $\lhd^{-1}$. The context $\dot{\mathcal{C}}'$ has a remote token, and the states $\dot{\mathcal{C}}'[\vec{H}^1]$ and $\dot{\mathcal{C}}'[\vec{H}^2]$ are both rooted. Therefore, we have $\dot{\mathcal{C}}'[\vec{H}^2]\ (\lhd^{\mathbb{C}})^{-1}\ \dot{\mathcal{C}}'[\vec{H}^1]$.

- When the token is a search token that is entering in $\dot{\mathcal{C}}$, because $\lhd$ is $(\mathbb{C}, Q^{-1}, Q')$-input-safe, we have one of the following three situations corresponding to (I), (II) and (III) of Def. 4.3.7.

  - There exist two stuck states $\dot{N}_1$ and $\dot{N}_2$ such that $\dot{G}_p \to^* \dot{N}_p$ for each $p \in \{1, 2\}$. By the determinism of transitions, we have $\dot{G}_2 \to \dot{G}'_2 \to^* \dot{N}_2$, which means the condition (I') is satisfied.

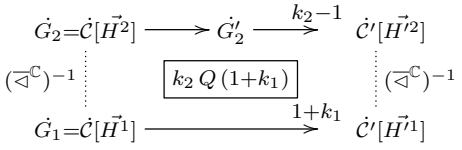
  - There exist a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ and two numbers $k_1, k_2 \in \mathbb{N}$, such that the token of $\dot{\mathcal{C}}'$ is not a rewrite token and not entering, $(1 + k_1)\ Q^{-1}\ k_2$, $\dot{\mathcal{C}}[\vec{H}^1] \to^{1+k_1} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}[\vec{H}^2] \to^{k_2} \dot{\mathcal{C}}'[\vec{H'}^2]$. We have

the following situation, namely the black part of the diagram below.

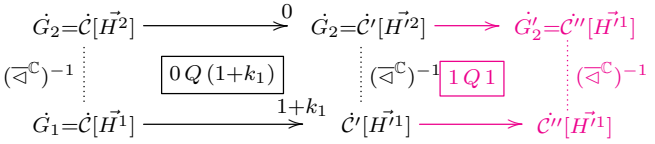



The magenta part holds, because the token of $\dot{\mathcal{C}}'$ is not a rewrite token and not entering, and because states $\dot{\mathcal{C}}'[\vec{H'}^1]$ and $\dot{\mathcal{C}}'[\vec{H'}^2]$ are rooted. We check the condition (II') by case analysis on the number $k_2$.

* When $k_2 > 0$, by the determinism of transitions, we have the following diagram, which means the condition (II') is fulfilled.



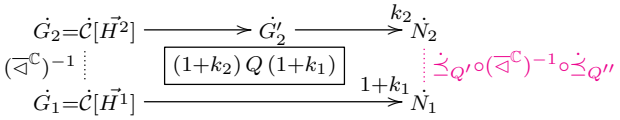

* When $k_2 = 0$, $\dot{G}_2 = \dot{\mathcal{C}}[\vec{H}^2] = \dot{\mathcal{C}}'[\vec{H'}^2]$, and we have the following situation, namely the black part of the diagram below.




Because $\dot{G}_2 \to \dot{G}'_2$, and the token of $\dot{\mathcal{C}}'$ is a value token, or a non-entering search token, by Lem. 4.4.5(3), there exists a focussed context $\dot{\mathcal{C}}''$ with a remote token, such that $|\dot{\mathcal{C}}''| = |\dot{\mathcal{C}}'|$ and $\dot{\mathcal{C}}'[\vec{H'}^p] \to \dot{\mathcal{C}}''[\vec{H'}^p]$ for each $p \in \{1, 2\}$. By the determinism of transitions, $\dot{G}'_2 = \dot{\mathcal{C}}''[\vec{H'}^1]$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q$ is a preorder and $1\ Q\ 1$. The context $\dot{\mathcal{C}}''$ satisfies $|\dot{\mathcal{C}}''| = |\dot{\mathcal{C}}'| \in \mathbb{C}$, so $(\dot{\mathcal{C}}''; \vec{H'}^2; \vec{H'}^1)$ is a $\mathbb{C}$-specimen of $\lhd^{-1}$. The context $\dot{\mathcal{C}}''$ has a remote token, and the states $\dot{\mathcal{C}}''[\vec{H'}^1]$ and $\dot{\mathcal{C}}''[\vec{H'}^2]$ are both rooted. Therefore, we have $\dot{\mathcal{C}}''[\vec{H'}^2]\ (\overline{\lhd}^{\mathbb{C}})^{-1}\ \dot{\mathcal{C}}''[\vec{H'}^1]$. Finally, because $(Q, Q', Q'')$ is a rea-

sonable triple, $Q$ is closed under addition, and hence $1\ Q\ (2 + k_1)$. The condition (II') is fulfilled.

– There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_1, \dot{N}_2)$ of $\lhd$ up to $(\dot{\simeq}_{Q'}, \dot{\simeq}_{Q'})$, whose token is not a rewrite token, and two numbers $k_1, k_2 \in \mathbb{N}$, such that $(1 + k_1)\ Q^{-1}\ (1 + k_2)$, $\dot{\mathcal{C}}[\vec{H}^1] \rightarrow^{1+k_1} \dot{N}_1$, and $\dot{\mathcal{C}}[\vec{H}^2] \rightarrow^{1+k_2} \dot{N}_2$. By the determinism of transitions, we have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II') is fulfilled.




Because $(\dot{N}_1, \dot{N}_2)$ is a quasi-$\mathbb{C}$-specimen of $\lhd$ up to $(\dot{\simeq}_{Q'}, \dot{\simeq}_{Q'})$, and states $\dot{N}_1$ and $\dot{N}_2$ are rooted, there exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'}^1; \vec{H'}^2)$ of $\lhd$ with a non-rewrite token, such that $\dot{\mathcal{C}}'[\vec{H'}^1]$ and $\dot{\mathcal{C}}'[\vec{H'}^2]$ are also rooted, $\dot{N}_1 \dot{\simeq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^1]$, and $\dot{\mathcal{C}}'[\vec{H'}^2] \dot{\simeq}_{Q'} \dot{N}_2$. Because $(Q, Q', Q'')$ is a reasonable triple, $Q' \subseteq Q''$, and hence $\dot{\simeq}_{Q'} \subseteq \dot{\simeq}_{Q''}$. Therefore, we have:

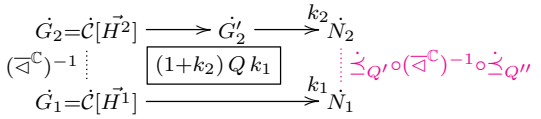
$$\dot{N}_2 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'}^2]\ (\overline{\lhd}^{\mathbb{C}})^{-1}\ \dot{\mathcal{C}}'[\vec{H'}^1] \dot{\preceq}_{Q''} \dot{N}_1.$$

- When the token is a rewrite token, $\dot{G}_2 \rightarrow \dot{G}'_2$ is a rewrite transition, and by definition of contextual lifting, the token is not entering in $\dot{\mathcal{C}}$. Because $\lhd^{-1}$ is $(\mathbb{C}, Q, Q', Q'')$-robust relative to all rewrite transitions, and $\dot{G}_1$ and $\dot{G}_2$ are rooted, we have one of the following two situations corresponding to (II) and (III) of Def. 4.3.11.

  – There exists a stuck state $\dot{N}$ such that $\dot{G}'_2 \rightarrow^* \dot{N}$. The condition (I') is satisfied.

  – There exist a quasi-$\mathbb{C}$-specimen $(\dot{N}_2, \dot{N}_1)$ of $\lhd^{-1}$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, whose

token is not a rewrite token, and two numbers $k_2, k_1 \in \mathbb{N}$, such that $(1 + k_2)\ Q\ k_1$, $\dot{G}'_2 \to^{k_2} \dot{N}_2$, and $\dot{G}_1 \to^{k_1} \dot{N}_1$. We have the following situation, namely the black part of the diagram below. Showing the magenta part confirms that the condition (II') is fulfilled.




Because $(\dot{N}_2, \dot{N}_1)$ is a quasi-$\mathbb{C}$-specimen of $\lhd^{-1}$ up to $(\dot{\preceq}_{Q'}, \dot{\preceq}_{Q''})$, and states $\dot{N}_2$ and $\dot{N}_1$ are rooted, there exists a $\mathbb{C}$-specimen $(\dot{\mathcal{C}}'; \vec{H'^2}; \vec{H'^1})$ of $\lhd^{-1}$ with a non-rewrite token, such that $\dot{\mathcal{C}}'[\vec{H'^2}]$ and $\dot{\mathcal{C}}'[\vec{H'^1}]$ are also rooted, $\dot{N}_2 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'^2}]$, and $\dot{\mathcal{C}}'[\vec{H'^1}] \dot{\preceq}_{Q''} \dot{N}_1$. This means $\dot{\mathcal{C}}'[\vec{H'^2}]\ \overline{\lhd^{-1}}^{\mathbb{C}}\ \dot{\mathcal{C}}'[\vec{H'^1}]$, and hence:

$$\dot{N}_2 \dot{\preceq}_{Q'} \dot{\mathcal{C}}'[\vec{H'^2}]\ (\overline{\lhd}^{\mathbb{C}})^{-1}\ \dot{\mathcal{C}}'[\vec{H'^1}] \dot{\preceq}_{Q''} \dot{N}_1.$$

□

## 4.5 Applications of the characterisation theorem

This section shows applications of Thm. 4.3.14, with respect to an instantiation $\mathcal{U}(\mathbb{O}^{\text{ex}}, B_{\mathbb{O}^{\text{ex}}})$ of the universal abstract machine. We start by defining the specific operation set $\mathbb{O}^{\text{ex}}$ and its behaviour $B_{\mathbb{O}^{\text{ex}}}$ in Sec. 4.5.1, which are informally introduced in Sec. 3.4.2, and proceed to illustrate proofs of the observational equivalences listed in Sec. 3.4. Necessary templates and their robustness are discussed in Sec. 4.5.2, and Sec. 4.5.3 describes how these templates can be combined to prove the observational equivalences. Finally, Sec. 4.5.4 and Sec. 4.5.5 give further details of the reasoning about templates and their robustness.

### 4.5.1 Properties of compute transitions

The operation set $\mathbb{O}^{\mathrm{ex}} = \mathbb{O}^{\mathrm{ex}}_{\checkmark} \uplus \mathbb{O}^{\mathrm{ex}}_{\lightning}$ we use here is taken from those discussed in Sec. 3.4.2. It is namely given by passive operations $\mathbb{O}^{\mathrm{ex}}_{\checkmark} = \mathbb{Z} \cup \{\lambda, (), \mathtt{tt}, \mathtt{ff}\}$, where $\mathbb{Z}$ is the set of integers, and active operations $\mathbb{O}^{\mathrm{ex}}_{\lightning} = \{\vec{@}, \mathtt{ref}, =, :=, !, +, -, -_1\}$.

The behaviour $B_{\mathbb{O}^{\mathrm{ex}}}$, namely compute transitions for the active operations $\mathbb{O}^{\mathrm{ex}}_{\lightning}$, are all defined locally via rewrite rules; for function application in Fig. 3.7, reference manipulation in Fig. 3.9 and Fig. 3.10, and arithmetic in Fig. 3.8 (showing just addition, but other operators can be similarly added). In these rules, the hypernet $G_S$ is additionally required to be stable.

Determinism and refocusing of the particular machine $\mathcal{U}(\mathbb{O}^{\mathrm{ex}}, B_{\mathbb{O}^{\mathrm{ex}}})$ boils down to determinism and preservation of the rooted property of compute transitions.

Compute transitions of operations $\{\vec{@}, \mathtt{ref}, +, -, -_1\}$ are deterministic, because at most one rewrite rule can be applied to each state. In particular, the stable hypernet $G_S$ in the figures is uniquely determined (by Lem. A.4.1(3)).

As discussed in Sec. 3.5, copy transitions are all deterministic, because any possible application of a contraction rule results in the same state. Compute transitions of name-accessing operations $\{=, :=, !\}$ are deterministic for the same reason.

Compute transitions of all the operations $\mathbb{O}^{\mathrm{ex}}_{\lightning}$ are stationary, and hence they preserve the rooted property. The stationary property can be checked using local rewrite rules. Namely, in each rewrite rule $\dot{H} \mapsto \dot{H}'$ of the operations, only one input of $|\dot{H}|$ has type $\star$, and $\dot{H} = \lightning; |\dot{H}|$ and $\dot{H}' = ?; |\dot{H}'|$. Moreover, any output of $|\dot{H}|$ with type $\star$ is a target of an atom edge or a box edge (by definition of stable hypernets), which implies $|\dot{H}|$ is one-way.

Because any initial state is rooted, given that all transitions preserve the rooted property, we can safely assume that any state that arises in an execution is rooted. This means that the additional specification on stable hypernets in local rewrite rules is in fact guaranteed to be satisfied in any execution (by Lem. A.3.5, Lem. A.4.6

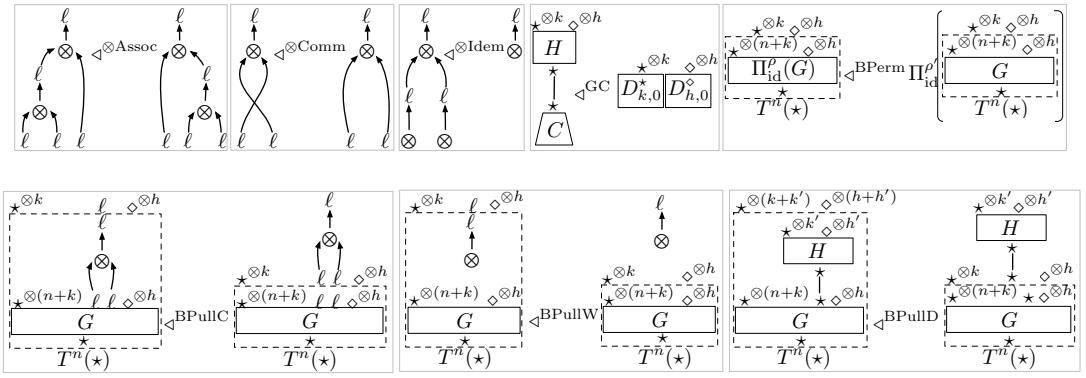



Figure 4.1: Structural pre-templates ($C : \epsilon \Rightarrow \star$ is a contraction tree, $H$ is a copyable hypernet, $G$ is a hypernet, $(\rho, \rho')$ is a box-permutation pair)

and Lem. A.4.4).

### 4.5.2 Example templates

Now we enumerate pre-templates that we will use to prove the equivalences listed in Sec. 3.4. We use the following auxiliary definition to specify one of these pre-templates (namely $\lhd^{\mathrm{BPerm}}$ in Fig. 4.1).

**Definition 4.5.1** (Box-permutation pair)**.** For any $n, k, h \in \mathbb{N}$, let $\rho$ and $\rho'$ be bijections on sets $\{1, \ldots, n+k+h\}$ and $\{1, \ldots, k+h\}$, respectively. These bijections form a *box-permutation* pair $(\rho, \rho')$ if, for each $i \in \{1, \ldots, n+k+h\}$, the following holds:

(A) $\rho(i) = i$ if $1 \leq i \leq n$,

(B) $\rho(i) = \rho'(i-n)$ if $n < i \leq n+k+h$,

(C) $1 \leq \rho'(i-n) \leq k$ if $n < i \leq n+k$,

(D) $k < \rho'(i-n) \leq k+h$ if $n+k < i \leq n+k+h$.

The pre-templates we use are classified into three: *structural* pre-templates, *operational* pre-templates, and *name-exhaustive* pre-templates. While the structural laws in Sec. 3.4.1 can be proved using only structural pre-templates, the beta laws

and stateful laws in Sec. 3.4.2 require operational and name-exhaustive pre-templates as well as structural pre-templates.

Fig. 4.1 shows all the structural pre-templates but the one derived from contraction rules: namely, $|\dot{G}_1| \lhd^{\otimes} |\dot{G}_2|$ whenever $\dot{G}_1 \mapsto \dot{G}_2$ is a contraction rule. The structural pre-templates primarily concern contraction edges, weakening edges and box edges. Contextual equivalences implied by $\lhd^{\otimes\mathrm{Assoc}}$, $\lhd^{\otimes\mathrm{Comm}}$ and $\lhd^{\otimes\mathrm{Idem}}$ enable the so-called idempotent completion (aka. Karoubi envelope or Cauchy completion) on contractions and weakenings. This means that contraction trees with the same type can be identified, so long as they contain at least one weakening edge.

We use two operational pre-templates, which are directly derived from some local rewrite rules of active operations. Namely, $|\dot{G}_1| \lhd^{\vec{@}} |\dot{G}_2|$ if $\dot{G}_1 \mapsto \dot{G}_2$ is a beta rewrite rule (Fig. 3.7); and $|\dot{G}_1| \lhd^{\mathtt{ref}} |\dot{G}_2|$ if $\dot{G}_1 \mapsto \dot{G}_2$ is a reference-creation rewrite rule (Fig. 3.9). Note that we keep the additional specification that $G_S$ in these figures are stable hypernets.

Fig. 4.2 shows the last class of pre-templates, namely four name-exhaustive pre-templates. They are specific to the four stateful laws in Desiderata 3.4.5, and they analyse possible usages of a single name of interest.

Output-closure of all the pre-templates can be easily checked, typically by spotting that an input or an output, of type $\star$, is a source or a target of a contraction, atom or box edge.

Table. 4.1 outlines the way we will use Thm. 4.3.14 on all the pre-templates. For example, $\lhd^{\otimes}$ is a $(\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}, \geq_{\mathbb{N}}, =_{\mathbb{N}})$-template, as shown in the "template" column, and both itself and its converse are $(\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}, =_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$-robust relative to all rewrite transitions, as shown in the "robustness" columns. Thanks to the monotonicity (Remark 4.3.15), we can use Thm. 4.3.14(1) with a reasonable triple $(\geq_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$, and Thm. 4.3.14(2) with a reasonable triple $(\leq_{\mathbb{N}}, =_{\mathbb{N}}, =_{\mathbb{N}})$. Consequently, $H_1 \lhd^{\otimes} H_2$ implies $H_1 \preceq^{\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}}_{\geq_{\mathbb{N}}} H_2$ and $H_2 \preceq^{\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}}_{\leq_{\mathbb{N}}} H_1$, which is shown in the "implication of $H_1 \lhd H_2$" column.

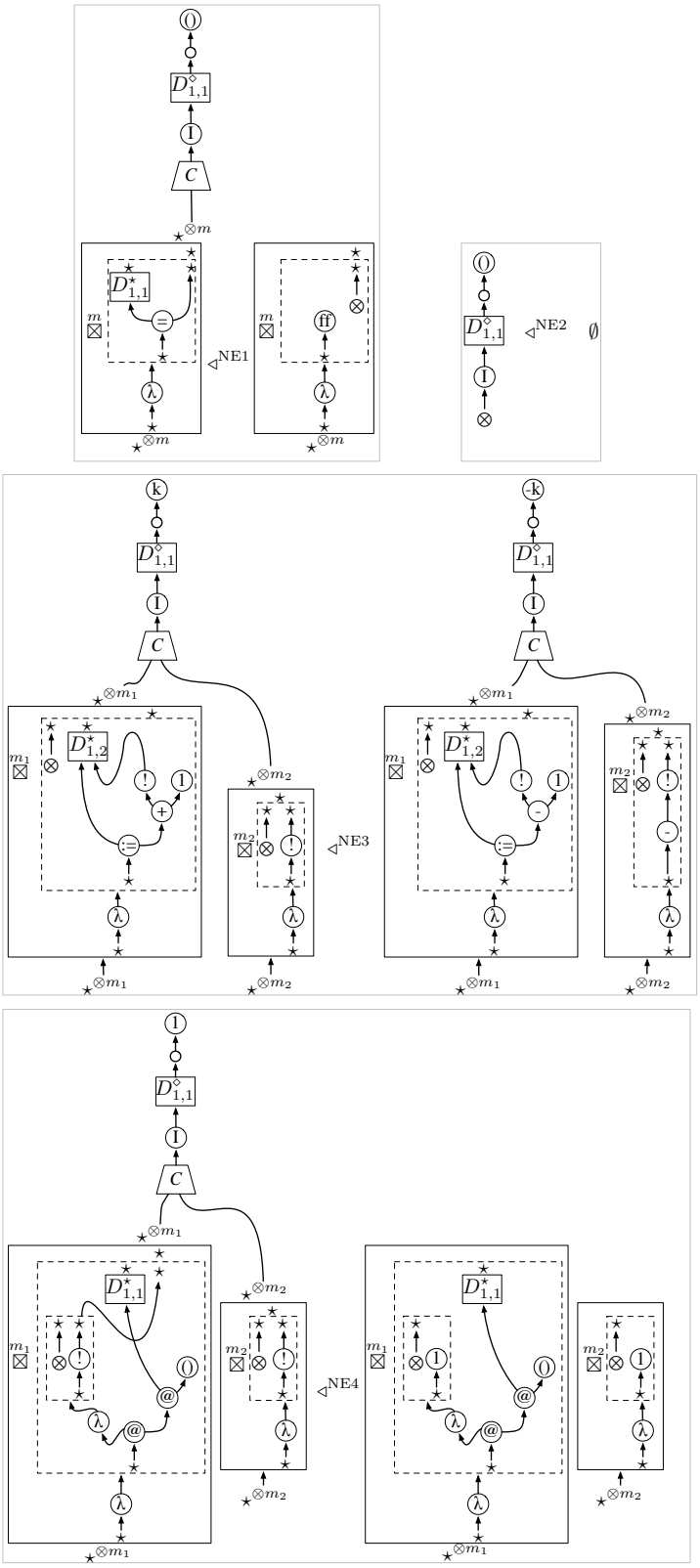



Figure 4.2: Name-exhaustive pre-templates ($C$ is a contraction tree)

| | template (input-safety) | robustness of $\lhd$ | robustness of $\lhd^{-1}$ | dependency | implication of $H_1 \lhd H_2$ |
|---|---|---|---|---|---|
| $\lhd^{\otimes\text{Assoc}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\otimes\text{Comm}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\otimes\text{Idem}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\otimes}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Comm}}$ | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\geq_{\mathbb{N}}} H_2$, $H_2 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\leq_{\mathbb{N}}} H_1$ |
| $\lhd^{\text{GC}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{BPerm}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{BPullC}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Comm}}$ $\lhd^{\otimes\text{Idem}}$ | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{BPullW}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\lhd^{\otimes\text{Idem}}$ | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{BPullD}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, \leq$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, \geq, =$ | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Comm}}$ $\lhd^{\otimes\text{Idem}}$ $\lhd^{\otimes}$ $\lhd^{\text{GC}}$ | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\leq_{\mathbb{N}}} H_2$, $H_2 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\geq_{\mathbb{N}}} H_1$ |
| | | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \leq, =, \leq$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, \geq, \geq$ | | |
| $\lhd^{\vec{@}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, =, =, =$ | — | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}}_{\geq_{\mathbb{N}}} H_2$, $H_2 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}}_{\leq_{\mathbb{N}}} H_1$ |
| | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, \geq, =$ | | | | |
| $\lhd^{\text{ref}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, =, =, =$ | — | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}}_{\geq_{\mathbb{N}}} H_2$, $H_2 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}}_{\leq_{\mathbb{N}}} H_1$ |
| | $\mathbb{C}_{\mathbb{0}^{\text{ex}\text{-bf}}}, \geq, =$ | | | | |
| $\lhd^{\text{NE1}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, =, =$ | — | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Idem}}$ $\lhd^{\text{GC}}$ | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\geq_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{NE2}}$ | $\square, \square, \square$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =, =$ | — | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{=_{\mathbb{N}}} H_2$ |
| $\lhd^{\text{NE3}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \leq, \geq, \leq$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, \geq, \leq$ | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Idem}}$ $\lhd^{\otimes}$ $\lhd^{\text{GC}}$ | $H_1 \simeq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\mathbb{N}\times\mathbb{N}} H_2$ |
| | | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \mathbb{N}\times\mathbb{N}, \geq, \mathbb{N}\times\mathbb{N}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \mathbb{N}\times\mathbb{N}, \geq, \mathbb{N}\times\mathbb{N}$ | | |
| $\lhd^{\text{NE4}}$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, =, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, \geq, =$ | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \leq, =, \leq$ | $\lhd^{\otimes\text{Assoc}}$ $\lhd^{\otimes\text{Idem}}$ $\lhd^{\otimes}$ $\lhd^{\text{GC}}$ | $H_1 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\geq_{\mathbb{N}}} H_2$, $H_2 \preceq^{\mathbb{C}_{\mathbb{0}^{\text{ex}}}}_{\leq_{\mathbb{N}}} H_1$ |
| | | $\mathbb{C}_{\mathbb{0}^{\text{ex}}}, \geq, \geq, \geq$ | | | |

Table 4.1: Templates, with their robustness and implied contextual refinements/equivalences ($\square$ denotes anything)

Pre-templates that relate hypernets with no input of type $\star$ are trivially a $(\mathbb{C}, Q, Q')$-template for any $\mathbb{C}$, $Q$ and $Q'$. The table uses '$\Box, \Box, \Box$' to represent this situation.

For many pre-templates, a reasonable triple can be found by selecting *bigger* parameters from those of input-safety and robustness, thanks to the monotonicity. However, pre-templates $\lhd^{\mathrm{BPullD}}$, $\lhd^{\mathrm{NE3}}$ and $\lhd^{\mathrm{NE4}}$ require non-trivial use of the monotonicity. For each of these pre-templates, an upper row shows a parameter $(\mathbb{C}, Q_1, Q_1', Q_1'')$ that makes it (or its converse) robust, and a lower row shows a parameter $(\mathbb{C}, Q_2, Q_2', Q_2'')$ to which Thm. 4.3.14 can be applied.

In the table, cyan symbols indicate where a proof of input-safety or robustness relies on contextual refinement. The "dependency" column indicates which pre-templates can be used to prove the necessary contextual refinement, given that these pre-templates imply contextual refinement as shown elsewhere in the table. This reliance specifically happens in finding a quasi-specimen, using contextual refinements/equivalences via Lem. 4.3.19. In the case of $\lhd^{\otimes}$, its input-safety and robustness are proved under the assumption that $\lhd^{\otimes\mathrm{Assoc}}$ and $\lhd^{\otimes\mathrm{Comm}}$ imply contextual equivalence $\simeq^{\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}}_{=_{\mathbb{N}}}$.

The restriction to binding-free contexts plays a crucial role only in robustness regarding the operational pre-templates $\lhd^{\vec{@}}$ and $\lhd^{\mathtt{ref}}$. In fact, these pre-templates are input-safe with respect to both $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$ and $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}\text{-}\mathrm{bf}}$. This gap reflects duplication behaviour on atom edges, which is only encountered in a proof of robustness. A shallow atom edge is never duplicated, whereas a deep one can be duplicated as part of a box (which represents a thunk).

Finally, the pre-template $\lhd^{\mathrm{NE1}}$ is the only example whose converse is not robust. This is because the equality operation '=' is only defined on names, whereas it is possible in Spartan to give values, other than names, as arguments of the equality operation.

The key part of proving input-safety or robustness of a pre-template is to analyse

| | ⊗ Assoc | ⊗ Comm | ⊗ Idem | ⊗ | GC | BPerm | BPullC | BPullW | BPullD | ↑@ | `ref` | NE$i$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Weakening | | | ○ | | | | | ○ | | | | |
| Exchange | | | | | | ○ | | | | | | |
| Struct. (3.3) (3.4) | ○ | ○ | ○, (W) | | | (Ex) | | (W) | | | | |
| Struct. (3.5) (3.6) | ○ | ○ | ○, (W) | | | | | (W) | | | | |
| Aux. Copy | ○ | ○ | ○ | ○ | ○ | | | | | | | |
| Aux. Subst. | ● | ● | ● | ● | ● | ○ | ○ | ○ | ○ | | | |
| Struct. (3.7) | ○ | ○ | ○, (W) | ○ | ○ | | | (W) | | | | |
| Struct. (3.8) | ○ | ○ | ○, (W) | ○ | ○ | ○, (Ex) | ○ | ○, (W) | ○ | | | |
| Micro beta | | ○ | | | | (Ex) | | | | ○ | | |
| Freshness | ○ | ○ | ○, (W) | | ● | ○ | ○ | ○, (W) | | ○ | ○ | 1 |
| Locality | ○ | ○ | ○, (W) | | | | | (W) | | ○ | ○ | 2 |
| Param. 1 | ○ | ○ | ○, (W) | ● | ● | ○ | ○ | ○, (W) | | ○ | ○ | 3 |
| Param. 2 | ○ | ○ | ○, (W) | ● | ● | ○ | ○ | ○, (W) | | ○ | ○ | 4 |

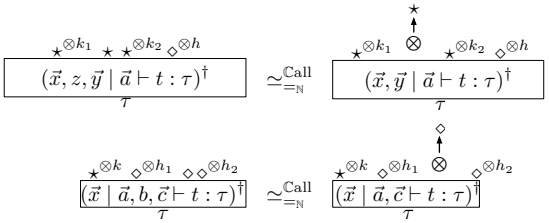
Table 4.2: Dependency of contextual refinements/equivalences on templates

how a rewrite transition involves edges (at any depth) of a state that are contributed by the pre-template. Given that all rewrite transitions (including copy transitions) are specified by local rewrite rules, the analysis boils down to checking possible overlaps between a local rule and the pre-template. A typical situation is where a local rule simply preserves or duplicates edges contributed by a pre-template, without breaking them. Lem. 4.5.2 in Sec. 4.5.4 identifies two such situations: when the overlaps are all about deep edges, and when the pre-template relates contraction trees only.

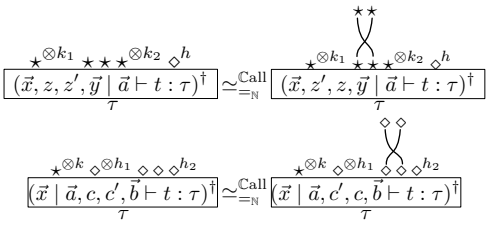
### 4.5.3 Combining templates

Each law in Sec. 3.4 can be proved for the operation sets $\mathbb{O}^{\mathrm{ex}}_{\checkmark} = \mathbb{Z} \cup \{\lambda, (), \mathtt{tt}, \mathtt{ff}\}$ and $\mathbb{O}^{\mathrm{ex}}_{\not z} = \{\vec{@}, \mathtt{ref}, =, :=, !, +, -, -_1\}$ by combining the pre-templates. The assumption here is that the pre-templates imply contextual refinement as listed in Table 4.1. We describe below how each law depends on pre-templates, using Table 4.2. The (full) Beta law is simply the combination of the Micro Beta law and the substitution law (3.8), so it is omitted in the table.

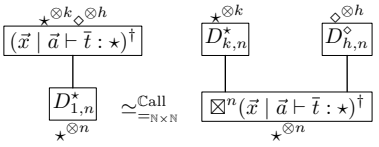
The table includes some extra laws that are useful to prove the laws in Sec. 3.4. These laws are namely: Weakening, Exchange, Auxiliary Copy and Auxiliary Substitution; and shown in Fig. 4.3. When a law can be proved using the Weakening



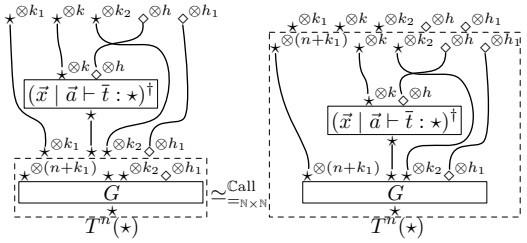

(a) Weakening ($z \notin fv(t)$, $b \notin fa(t)$)




(b) Exchange




(c) Auxiliary Copy ($\bar{t}$ is non-generative)




(d) Auxiliary Substitution ($\bar{t}$ is non-generative)

Figure 4.3: Auxiliary laws

or Exchange law, it is indicated by (W) or (E), respectively, in the table. The Auxiliary Copy law is used to prove both the Structural laws (3.7) and (3.8), while the Auxiliary Substitution law is used to prove the Structural law (3.8).

The symbol '$\circ$' indicates *direct* dependency on pre-templates, in the sense that a law can be proved by combining contextual refinements implied by these pre-templates. For these pre-templates to imply contextual refinements, some other pre-templates may need to imply contextual refinements; these other pre-templates are *indirectly* depended by the law. The indirect dependency is indicated by the

symbol '•'.

Thanks to the Weakening law, it suffices to check the minimum judgement to prove an observational equivalence. This is the case for binding-free contexts, too, because any context that consists of a hole with no target, and weakening edges, is binding-free. For example, Fig. 4.4 illustrates a proof of the Parametricity 2 law in the empty environment $(- \mid - \vdash \Box : \star)$, and this proof is enough to show the law in any environment $(\vec{x} \mid \vec{a} \vdash \Box : \star)$.

### 4.5.4 Local reasoning

For the particular operation set $\mathbb{O}^{\mathrm{ex}}$, the following lemma identifies two typical situations, where a local rule simply preserves or duplicates edges contributed by a pre-template, without breaking them.

**Lemma 4.5.2.** *Assume any $\mathbb{C}$-specimen of an output-closed pre-template $\lhd$ that has the form*

$$(\mathcal{C}_1[\vec{\chi}', \dot{\mathcal{C}}_2[\vec{\chi}'']]; \vec{G}', \vec{G}''; \vec{H}', \vec{H}''),$$

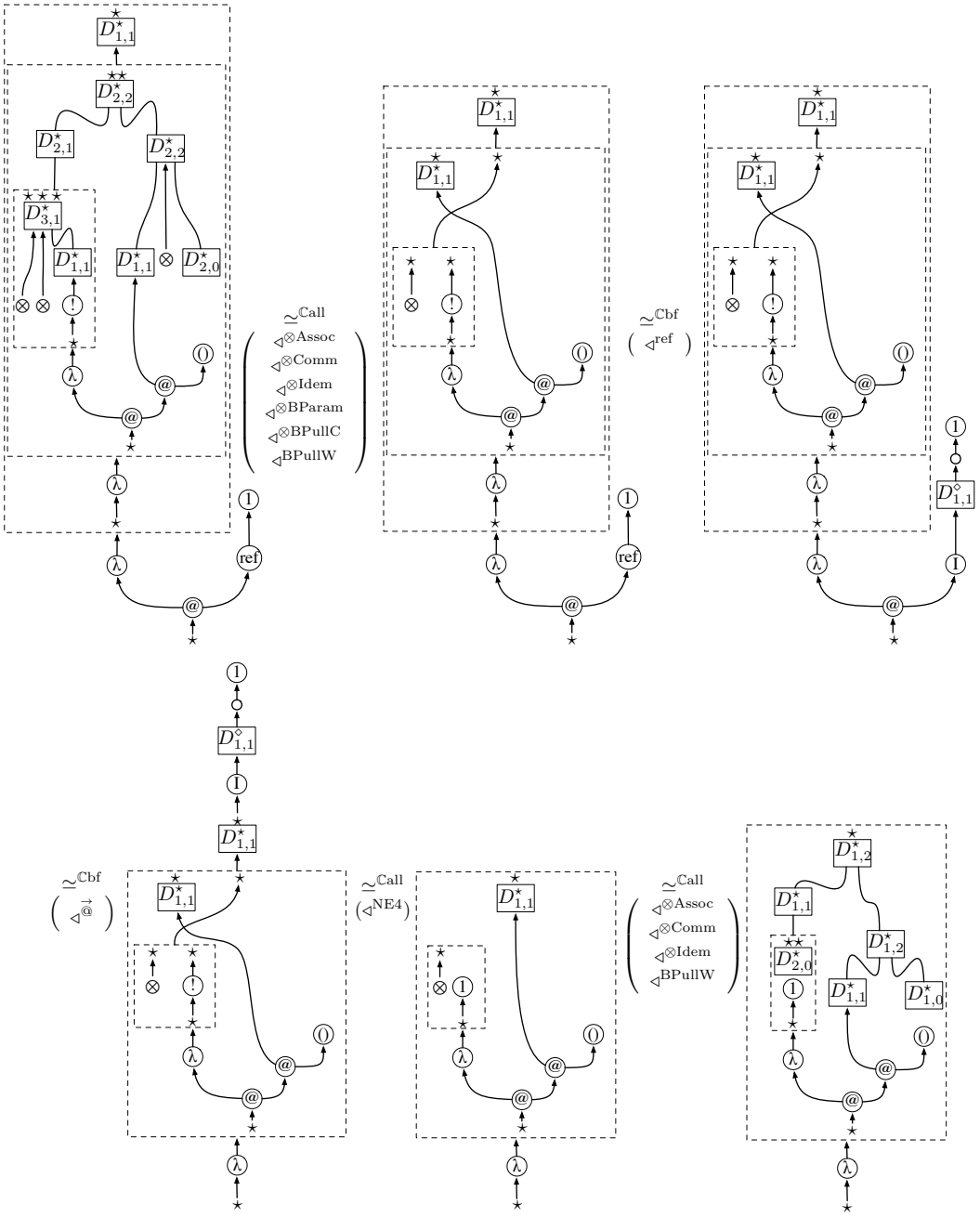
*and any focussed hypernet $\dot{N}$, such that $\mathcal{C}_1[\vec{G}', \dot{\mathcal{C}}_2[\vec{G}'']]$ and $\mathcal{C}_1[\vec{H}', \dot{\mathcal{C}}_2[\vec{H}'']]$ are states, and $\dot{\mathcal{C}}_2[\vec{G}''] \mapsto \dot{N}$ is a contraction rule or a local rewrite rule of an operation of $\mathbb{O}^{\mathrm{ex}}_{\ell} = \{\vec{@}, \mathtt{ref}, =, :=, !, +, -, -_1\}$.*

1. *If all holes of $\dot{\mathcal{C}}_2$ are deep, then there exist a focussed context $\dot{\mathcal{C}}'_2$ and two sequences $\vec{G}'''$ and $\vec{H}'''$ of focus-free hypernets, such that $\dot{\mathcal{C}}_2[\vec{G}''] \mapsto \dot{\mathcal{C}}'_2[\vec{G}'''] = \dot{N}$, $\dot{\mathcal{C}}_2[\vec{H}''] \mapsto \dot{\mathcal{C}}'_2[\vec{H}''']$, and $G'''_i \lhd H'''_i$ for each index $i$.*

2. *In the situation of (1), if additionally both states $\mathcal{C}_1[\vec{G}', \dot{\mathcal{C}}_2[\vec{G}'']]$ and $\mathcal{C}_1[\vec{H}', \dot{\mathcal{C}}_2[\vec{H}'']]$ are rooted, and $|\mathcal{C}_1[\vec{\chi}', \dot{\mathcal{C}}_2[\vec{\chi}'']]|$ is binding-free, then $\dot{\mathcal{C}}'_2$ can be taken so that $|\mathcal{C}_1[\vec{\chi}', \dot{\mathcal{C}}'_2[\vec{\chi}''']]|$ is also binding-free.*

3. *If the pre-template $\lhd$ is a relation on contraction trees, then there exist a focussed context $\dot{\mathcal{C}}'_2$ and two sequences $\vec{G}'''$ and $\vec{H}'''$ of focus-free hypernets,*

$(- \mid - \vdash (\lambda x.\lambda f.(\lambda z.!x)\vec{@}(f\vec{@}()))\vec{@}(\mathrm{ref}\,1) : \star)^\dagger =$




$= (- \mid - \vdash \lambda f.(\lambda z.1)\vec{@}(f\vec{@}()) : \star)^\dagger.$

Figure 4.4: A proof outline of the Parametricity 2 law, in the empty environment

*such that* $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}] = \dot{N}$, $\dot{\mathcal{C}}_2[\vec{H''}] \mapsto \dot{\mathcal{C}}'_2[\vec{H'''}]$, *and* $G'''_i \lhd H'''_i$ *for each index* $i$.

*Proof of the point (1).* The proof is by case analysis on the local rule $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{N}$.

- When the rule is a contraction rule, the rule simply duplicates all box edges without changing any deep edges. Because all holes of $\dot{\mathcal{C}}_2$ are deep, there exists a focussed context $\dot{\mathcal{C}}'_2$, whose holes are all deep, such that $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G''}, \vec{G''}]$ is a contraction rule. Moreover, $\dot{\mathcal{C}}_2[\vec{H''}] \mapsto \dot{\mathcal{C}}'_2[\vec{H''}, \vec{H''}]$ is also a contraction rule.

- When the rule is a beta rewrite rule, the rule replaces a box edge with the hypernet that labels the box. Because all holes of $\dot{\mathcal{C}}_2$ are deep, there exists a focussed context $\dot{\mathcal{C}}'_2$ such that $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G''}]$ is a beta rewrite rule. Moreover, $\dot{\mathcal{C}}_2[\vec{H''}] \mapsto \dot{\mathcal{C}}'_2[\vec{H''}]$ is also a beta rewrite rule.

- Otherwise, the rule does not involve any deep edge. This means the focussed context $\dot{\mathcal{C}}_2$ must have no hole, and the sequences $\vec{G''}$ and $\vec{H''}$ must be empty. We can take a focussed context $\dot{\mathcal{C}}'_2$ with no hole, such that $\dot{\mathcal{C}}_2[\,] \mapsto \dot{\mathcal{C}}'_2[\,]$.

Because contraction rules and local rewrite rules are all deterministic, $\dot{\mathcal{C}}'_2[\vec{G'''}] = \dot{N}$ follows from $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}]$. □

*Proof of the point (2).* The proof is built on top of the proof of the point (1). In particular, we assume that all holes of $\dot{\mathcal{C}}'_2$ are deep in the case of contraction rules, and $\dot{\mathcal{C}}'_2$ has no hole in the case of local rules of $\mathbb{O}_t^{\mathrm{ex}} \backslash \{\vec{@}\}$. Under this assumption, our goal is to prove that $|\mathcal{C}_1[\vec{\chi'}, \dot{\mathcal{C}}'_2]|$ is binding-free.

Let $\mathcal{C}$ denote $\mathcal{C}_1[\vec{\chi'}, |\dot{\mathcal{C}}_2|]$ and $\mathcal{C}'$ denote $\mathcal{C}_1[\vec{\chi'}, |\dot{\mathcal{C}}'_2|]$. Because $\mathcal{C} = |\mathcal{C}_1[\vec{\chi'}, \dot{\mathcal{C}}_2]|$ and $\mathcal{C}' = |\mathcal{C}_1[\vec{\chi'}, \dot{\mathcal{C}}'_2]|$, it suffices to prove that $\mathcal{C}'$ is binding-free, given that $\mathcal{C}$ is binding-free.

Firstly, because $\mathcal{C}$ is binding-free, $|\dot{\mathcal{C}}_2|$ is necessarily binding-free. By inspecting the local rule $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}]$, we check below that $|\dot{\mathcal{C}}'_2|$ is also binding-free.

- When the rule is a contraction rule, or a beta rewrite rule, any path that makes $|\dot{\mathcal{C}}_2'|$ not binding-free gives a path in $|\dot{\mathcal{C}}_2|$, which leads to a contradiction.
- Otherwise, $|\dot{\mathcal{C}}_2'|$ is trivially binding-free because it does not have any hole edges.

Now we prove that $\mathcal{C}'$ is binding-free by contradiction; we assume that there exists a path $P$ in $\mathcal{C}'$, from a source of a contraction, atom, box or hole edge $e$, to a source of a hole edge $e'$. Let $\chi$ be the last hole label of $\mathcal{C}_1$ (i.e. $\mathcal{C}_1[\vec{\chi'}, \chi]$), and $e_\chi$ denote the hole edge of $\mathcal{C}_1$ labelled with $\chi$. We derive a contradiction by case analysis on the path $P$.

- When $e'$ comes from $\mathcal{C}_1$, and the path $P$ consists of edges from $\mathcal{C}_1$ only, the path $P$ gives a path in $\mathcal{C}_1$. Because $P$ does not contain $e_\chi$, it also gives a path in $\mathcal{C}$. This contradicts $\mathcal{C}$ being binding-free.
- When $e'$ comes from $\mathcal{C}_1$, and the path $P$ contains an edge from $\mathcal{C}_2$, by finding the last edge from $\mathcal{C}_2$ in $P$, we can take a suffix of $P$ that gives a path $P'$ from a target of the hole edge $e_\chi$ to a source of the hole edge $e'$, in $\mathcal{C}_1$. Because the suffix path $P'$ does not contain $e_\chi$, it gives a path in $\mathcal{C}$. We inspect the local rule $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}_2'[\vec{G'''}]$ as follows.
  - When the rule is a contraction rule, because each output of $|\dot{\mathcal{C}}_2|$ is reachable from a source of a contraction edge, adding the contraction edge at the beginning of $P'$ gives a path in $\mathcal{C}$ from a source of the contraction edge to a source of the hole edge $e'$. This contradicts $\mathcal{C}$ being binding-free.
  - When the rule is a rewrite rule of operations $\{\vec{@}, \texttt{ref}, =, :=, !\}$, $P'$ gives a path in $\mathcal{C}$ to the hole edge $e'$, from either of the following: a target of an atom or box edge; or a source of a contraction, atom or a box edge (by typing). This contradicts $\mathcal{C}$ being binding-free.
  - Otherwise, i.e. when the rule is a rewrite rule of operations $\{+, -, -_1\}$, the hole edge $e_\chi$ actually does not have any target. This contradicts, in

the first place, the path $P$ containing an edge from $\mathcal{C}_2$.

- When both $e$ and $e'$ come from $|\dot{\mathcal{C}}'_2|$, and the path $P$ gives a path in $|\dot{\mathcal{C}}'_2|$, this contradicts $|\dot{\mathcal{C}}'_2|$ being binding-free.

- When both $e$ and $e'$ come from $|\dot{\mathcal{C}}'_2|$, and the path $P$ does not give a single path in $|\dot{\mathcal{C}}'_2|$, we inspect the local rule $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}]$ as follows.

  - When the rule is a contraction rule, all holes of $\dot{\mathcal{C}}'_2$ are deep. It is impossible for $P$, which is to a target of a deep hole edge, not to give a path in $|\dot{\mathcal{C}}'_2|$. This is a contradiction.

  - When the rule is a local rewrite rule of any operations but '$\vec{@}$', $\dot{\mathcal{C}}'_2$ actually does not have any hole edge. It is impossible for the hole edge $e'$ to be from $|\dot{\mathcal{C}}'_2|$. This is a contradiction.

  - When the rule is a beta rewrite rule, the hole edge $e_\chi$ actually has only one source. If $P$ consists of edges from $|\dot{\mathcal{C}}'_2|$ only, the source must be also a target of $e_\chi$; otherwise, $P$ has a sub-sequence that gives a path in $\mathcal{C}_1$. In either case, there exists a path $P'$ from the source of $e_\chi$ to the source of $e_\chi$, i.e. a cycle around the source of $e_\chi$, in $\mathcal{C}_1$. This cycle gives a cycle in $\mathcal{C}$ around the source of $e_\chi$.

    In the case of a beta rewrite rule, outputs are all reachable from the input, which coincides with the token source, in $\dot{\mathcal{C}}_2[\vec{\chi''}]$. Therefore, the cycle $P'$ also gives a cycle in $\mathcal{C}_1[\vec{\chi'}, \dot{\mathcal{C}}_2[\vec{\chi''}]]$, around the token source. Because $(\mathcal{C}_1[\vec{\chi'}, \dot{\mathcal{C}}_2[\vec{\chi''}]]; \vec{G'}, \vec{G''}; \vec{H'}, \vec{H''})$ is a $\mathbb{C}$-specimen of the output-closed pre-template $\lhd$, by Lem. A.3.7(3), at least one of the states $\mathcal{C}_1[\vec{G'}, \dot{\mathcal{C}}_2[\vec{G''}]]$ and $\mathcal{C}_1[\vec{H'}, \dot{\mathcal{C}}_2[\vec{H''}]]$ is not rooted. This is a contradiction.

- When $e$ comes from $\mathcal{C}_1$ and $e'$ comes from $|\dot{\mathcal{C}}'_2|$, we inspect the local rule $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}]$ as follows.

– When the rule is a contraction rule, by finding the last edge from $\mathcal{C}_1$ in $P$, we can take a suffix of $P$ that gives a path $P'$ from an input to a source of the hole edge $e'$ in $|\dot{\mathcal{C}}'_2|$. However, the path $P'$ cannot exist because the hole edge $e'$ is deep in $|\dot{\mathcal{C}}'_2|$. This is a contradiction.

– When the rule is a local rewrite rule of any operations but '$\vec{@}$', $\dot{\mathcal{C}}'_2$ actually does not have any hole edge. It is impossible for the hole edge $e'$ to be from $|\dot{\mathcal{C}}'_2|$. This is a contradiction.

– When the rule is a beta rewrite rule, by finding the first edge from $|\dot{\mathcal{C}}'_2|$ in $P$, we can take a prefix of $P$ that gives a path $P'$ from a source of the edge $e$ to a source of the hole edge $e_\chi$, in $\mathcal{C}_1$. Because $P$ does not contain $e_\chi$, $P'$ does not contain $e_\chi$ either.

In the case of a beta rewrite rule, $e_\chi$ has only one source, and the input of $\dot{\mathcal{C}}_2[\vec{\chi}'']$ coincides with the token source. Therefore, $P'$ gives a path from a source of $e$ to the token source in $\mathcal{C}_1[\vec{\chi}', \dot{\mathcal{C}}_2[\vec{\chi}'']]$, and this path is not an operation path, because of its first edge $e$. Because $(\mathcal{C}_1[\vec{\chi}', \dot{\mathcal{C}}_2[\vec{\chi}'']]; \vec{G}', \vec{G}''; \vec{H}', \vec{H}'')$ is a $\mathbb{C}$-specimen of the output-closed pre-template $\lhd$, by Lem. A.3.7(3), at least one of the states $\mathcal{C}_1[\vec{G}', \dot{\mathcal{C}}_2[\vec{G}'']]$ and $\mathcal{C}_1[\vec{H}', \dot{\mathcal{C}}_2[\vec{H}'']]$ is not rooted. This is a contradiction.

□

*Proof of the point (3).* The proof is by case analysis on the local rule $\dot{\mathcal{C}}_2[\vec{G}''] \mapsto \dot{N}$.

- When the rule is a contraction rule, the rule simply duplicates all box edges without changing any deep edges, and does not change any existing shallow contraction/weakening edges. Because $\lhd$ relates only contraction trees, deep edges from $\vec{G}''$ are duplicated and shallow ones from $\vec{G}''$ are preserved, by the rule. There exist a sequence $\vec{G}'''$ of contraction trees and a focussed context $\dot{\mathcal{C}}'_2$, such that $\vec{G}''' \subseteq \vec{G}''$ (as sets) and $\dot{\mathcal{C}}_2[\vec{G}''] \mapsto \dot{\mathcal{C}}'_2[\vec{G}'', \vec{G}''']$ is a contraction rule. Moreover, there exists a sequence $\vec{H}'''$ of contraction trees that satisfies

$\vec{H'''} \subseteq \vec{H''}$ (as sets) and corresponds to $\vec{G'''}$. Because $\lhd$ relates only contraction trees, $\dot{\mathcal{C}}_2[\vec{H''}] \mapsto \dot{\mathcal{C}}'_2[\vec{H''}, \vec{H'''}]$ is also a contraction rule.

- When the rule is a beta rewrite rule, the rule involves no shallow contraction trees. Because $\lhd$ relates only contraction trees, all holes of $\dot{\mathcal{C}}_2$ must be deep, and the proof is reduced to the point (1).

- When the rule is a rewrite rule of name-accessing operations $\{=, :=, !\}$, the rule preserves contraction trees at any depth. There exists a focussed context $\dot{\mathcal{C}}'_2$ such that $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G''}]$ is a local rewrite rule. Moreover, because $\lhd$ relates only contraction trees, $\dot{\mathcal{C}}_2[\vec{H''}] \mapsto \dot{\mathcal{C}}'_2[\vec{H''}]$ is also a local rewrite rule.

- Otherwise, the rule involves no contraction trees, which means the focussed context $\dot{\mathcal{C}}_2$ must have no hole, and the sequences $\vec{G''}$ and $\vec{H''}$ must be empty. We can take a focussed context $\dot{\mathcal{C}}'_2$ with no hole, such that $\dot{\mathcal{C}}_2[\,] \mapsto \dot{\mathcal{C}}'_2[\,]$.

Because contraction rules and local rewrite rules are all deterministic, $\dot{\mathcal{C}}'_2[\vec{G'''}] = \dot{N}$ follows from $\dot{\mathcal{C}}_2[\vec{G''}] \mapsto \dot{\mathcal{C}}'_2[\vec{G'''}]$. □

### 4.5.5 Details of input-safety and robustness proofs

In this section we give some details of proving input-safety and robustness of the pre-templates.

Fig. 4.5 lists triggers that we use to prove some input-safety and robustness of the pre-templates. Table 4.3 shows contextual refinements/equivalences implied by these triggers, given that some pre-templates (shown in the "dependency" column) imply contextual refinement as shown in Table 4.1. All the implications can be proved simply using the congruence property and transitivity of contextual refinement. Table 4.3 shows which pre-template requires each trigger in its proof of input-safety or robustness (in the "used for" column). Note that the converse of any trigger is again a trigger.

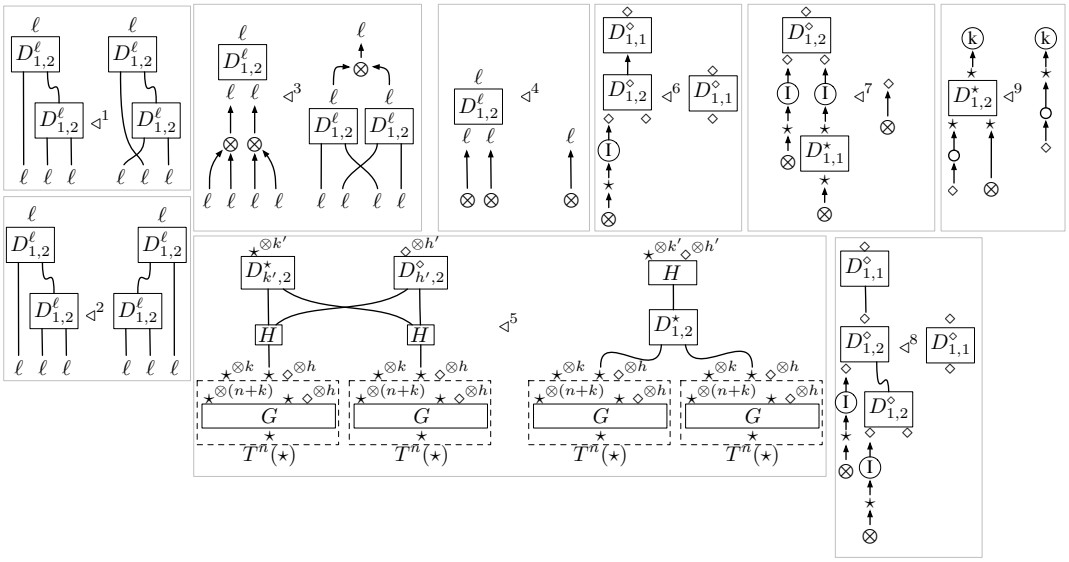


Figure 4.5: Triggers ($H$ is a copyable hypernet, and $G$ is a hypernet)

| | dependency | implication of $H_1 \lhd H_2$ | used for |
|---|---|---|---|
| $\lhd^1$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Comm}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\otimes}$ |
| $\lhd^2$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Comm}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\otimes}$ |
| $\lhd^3$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Comm}}$, $\lhd^{\otimes\text{Idem}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\text{BPullC}}$ |
| $\lhd^4$ | $\lhd^{\otimes\text{Idem}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\text{BPullW}}$, $\lhd^{\text{NE3}}$ |
| $\lhd^5$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Comm}}$, $\lhd^{\otimes\text{Idem}}$, $\lhd^{\otimes}$, $\lhd^{\text{GC}}$ | $H_1 \preceq^{\mathbb{C}^{\text{ex}}_{\leq\mathbb{N}}}_{\mathbb{O}} H_2$, $H_2 \preceq^{\mathbb{C}^{\text{ex}}_{\geq\mathbb{N}}}_{\mathbb{O}} H_1$ | $\lhd^{\text{BPullD}}$ |
| $\lhd^6$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Idem}}$, $\lhd^{\text{GC}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\text{NE1}}$, $\lhd^{\text{NE4}}$ |
| $\lhd^7$ | $\lhd^{\otimes\text{Idem}}$, $\lhd^{\otimes\text{GC}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\text{NE1}}$ |
| $\lhd^8$ | $\lhd^{\otimes\text{Assoc}}$, $\lhd^{\otimes\text{Idem}}$, $\lhd^{\text{GC}}$ | $H_1 \simeq^{\mathbb{C}^{\text{ex}}_{=\mathbb{N}}}_{\mathbb{O}} H_2$ | $\lhd^{\text{NE3}}$ |
| $\lhd^9$ | $\lhd^{\otimes}$, $\lhd^{\text{GC}}$ | $H_1 \preceq^{\mathbb{C}^{\text{ex}}_{\geq\mathbb{N}}}_{\mathbb{O}} H_2$, $H_2 \preceq^{\mathbb{C}^{\text{ex}}_{\leq\mathbb{N}}}_{\mathbb{O}} H_1$ | $\lhd^{\text{NE3}}$, $\lhd^{\text{NE4}}$ |

Table 4.3: Triggers and their implied contextual refinements/equivalences

Recall that there may be a choice of local rewrite rules to achieve the same copy transition, or the same compute transition of a name-accessing operation $\phi_{\lightning} \in \{=, :=, !\}$. This choice boils down to a choice of contraction trees. The minimum choice is to collect only contraction edges whose target is reachable from the token target. The maximum choice is to take the contraction tree(s) so that no contraction or weakening edge is incoming to the unique hole edge in a context.

**Pre-templates on contraction trees**

First we check input-safety and robustness of $\lhd^{\otimes\mathrm{Assoc}}$, $\lhd^{\otimes\mathrm{Comm}}$ and $\lhd^{\otimes\mathrm{Idem}}$, which are all on contraction trees.

Input-safety of $\lhd^{\otimes\mathrm{Assoc}}$ and $\lhd^{\otimes\mathrm{Comm}}$ can be checked as follows. Given a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen $(\dot{\mathcal{C}}; \vec{H}^1; \vec{H}^2)$ with an entering search token, because any input of a contraction tree is a source of a contraction edge, we have:

$$\dot{\mathcal{C}}[\vec{H}^1] \xrightarrow{\bullet} \langle \dot{\mathcal{C}}[\vec{H}^1] \rangle_{\lightning/?}, \quad \dot{\mathcal{C}}[\vec{H}^2] \xrightarrow{\bullet} \langle \dot{\mathcal{C}}[\vec{H}^2] \rangle_{\lightning/?}.$$

It can be observed that a rewrite transition is possible in $\langle \dot{\mathcal{C}}[\vec{H}^1] \rangle_{\lightning/?}$ if and only if a rewrite transition is possible in $\langle \dot{\mathcal{C}}[\vec{H}^2] \rangle_{\lightning/?}$. When a rewrite transition is possible in both states, we can use Lem. 4.5.2(3), by considering a maximal possible contraction rule. The results of the rewrite transition can be given by a new quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=, =)$ (here $=$ denotes equality on states). When no rewrite transition is possible, both of the states are not final but stuck.

Robustness of the three pre-templates and their converse can also be proved using Lem. 4.5.2(3), by considering a maximal possible local (contraction or rewrite) rule in each case.

**Input-safety of pre-templates not on contraction trees**

As mentioned in Sec. 4.5.2, pre-templates that relate hypernets with no input of type $\star$ are trivially input-safe for any parameter $(\mathbb{C}, Q, Q')$. This leaves us pre-templates $\lhd^{\otimes}$, $\lhd^{\vec{@}}$, $\lhd^{\mathtt{ref}}$, $\lhd^{\mathrm{NE1}}$, $\lhd^{\mathrm{NE3}}$ and $\lhd^{\mathrm{NE4}}$ to check.

As for $\lhd^{\otimes}$, note that the pre-template $\lhd^{\otimes}$ relates hypernets with at least one input. Any $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen of $\lhd^{\otimes}$ with an entering search token can be turned into the form $(\mathcal{C}[(?;_j \chi), \vec{\chi}]; H^1, \vec{H^1}; H^2, \vec{H^2})$ where $j$ is a positive number. The proof is by case analysis on the number $j$.

- When $j = 1$, we have:

$$\begin{aligned}\mathcal{C}[(?;_j H^1), \vec{H^1}] &\xrightarrow{\bullet} \mathcal{C}[(\lightning;_j H^1), \vec{H^1}] \\ &\to \mathcal{C}[(?;_j H^2), \vec{H^1}].\end{aligned}$$

  We can take $(\mathcal{C}[(?;_j H^2), \vec{\chi}]; \vec{H^1}; \vec{H^2})$ as a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen, and the token in $\mathcal{C}[(?;_j H^2), \vec{\chi}]$ is not entering.

- When $j > 1$, the token target must be a source of a contraction edge. There exist a focus-free context $\mathcal{C}'[\chi']$, two focus-free hypernets $H'^1 \lhd^{\otimes} H'^2$ and a focus-free hypernet $G$, such that

$$\begin{aligned}\mathcal{C}[(?;_j H^1), \vec{H^1}] &\xrightarrow{\bullet} \mathcal{C}[(\lightning;_j H^1), \vec{H^1}] \\ &\to \mathcal{C}[(?;_j \mathcal{C}'[H'^1]), \vec{H^1}], \\ \mathcal{C}[(?;_j H^2), \vec{H^2}] &\xrightarrow{\bullet} \mathcal{C}[(\lightning;_j H^2), \vec{H^2}] \\ &\to \mathcal{C}[(?;_j G), \vec{H^2}],\end{aligned}$$

  and $\mathcal{C}'[H'^2] \mathrel{\dot{\simeq}}_{=_{\mathbb{N}}} G$ given by the trigger $\lhd^1$ via Lem. 4.3.19. The results of these sequences give a quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=, \mathrel{\dot{\simeq}}_{=_{\mathbb{N}}})$.

A proof of input-safety of the operational pre-templates $\lhd^{\vec{@}}$ and $\lhd^{\mathtt{ref}}$ is a simpler

version of that of $\lhd^{\otimes}$, because the operational pre-templates relate hypernets with only one input.

Let $\mathbb{C}$ be either $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$ or $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}\text{-bf}}$. Any $\mathbb{C}$-specimen of an operational pre-template with an entering search token can be turned into the form $(\mathcal{C}[(?;\chi),\vec{\chi}];H^1,\vec{H^1};H^2,\vec{H^2})$. Note that the parameter $j$ that we had for $\lhd^{\otimes}$ is redundant in $?;\chi$. We have:

$$\begin{aligned}\mathcal{C}[(?;H^1),\vec{H^1}] &\xrightarrow{\bullet} \mathcal{C}[(\lightning;H^1),\vec{H^1}]\\ &\to \mathcal{C}[(?;H^2),\vec{H^1}].\end{aligned}$$

We can take $(\mathcal{C}[(?;H^2),\vec{\chi}];\vec{H^1};\vec{H^2})$ as a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen, and the token in $\mathcal{C}[(?;_j H^2),\vec{\chi}]$ is not entering. This data gives a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}\text{-bf}}$-specimen when $\mathbb{C}=\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}\text{-bf}}$, which follows from the closedness of $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}\text{-bf}}$ with respect to plugging (Lem. A.5.1). Note that $?;H^1$ can be seen as a context with no holes, which is trivially binding-free.

Finally, we look at the name-exhaustive pre-templates $\lhd^{\mathrm{NE1}}$, $\lhd^{\mathrm{NE3}}$ and $\lhd^{\mathrm{NE4}}$. Any $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen of one of these pre-templates, with an entering search token, can be turned into the form $(\mathcal{C}[(?;_j\chi),\vec{\chi}];H^1,\vec{H^1};H^2,\vec{H^2})$ where $j$ is a positive number. The token target is a source of an edge labelled with $\lambda\in\mathbb{O}^{\mathrm{ex}}_{\checkmark}$, so we have:

$$\begin{aligned}\mathcal{C}[(?;_j H^1),\vec{H^1}] &\xrightarrow{\bullet} \mathcal{C}[(\checkmark;_j H^1),\vec{H^1}],\\ \mathcal{C}[(?;_j H^2),\vec{H^2}] &\xrightarrow{\bullet} \mathcal{C}[(\checkmark;_j H^2),\vec{H^2}].\end{aligned}$$

The results of these sequences give a quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=,=)$.

**Robustness of pre-templates not on contraction trees: a principle**

Robustness can be checked by inspecting rewrite transition $\dot{\mathcal{C}}[\vec{H^1}]\to\dot{N}'$ from the state given by a specimen $(\dot{\mathcal{C}};\vec{H^1};\vec{H^2})$ of a pre-template, where the token of $\dot{\mathcal{C}}$ is not entering. We in particular consider the minimum local (contraction or rewrite) rule

$\dot{G} \mapsto \dot{G}'$ applied in this transition. This means that, in the hypernet $\dot{G}$, every vertex is reachable from the token target.

The inspection boils down to analysing how the minimum local rule involves edges that come from the hypernets $\vec{H}^1$. If all the involvement is deep, i.e. only deep edges from $\vec{H}^1$ are involved in the local rule, these deep edges must come via deep holes in the context $\dot{\mathcal{C}}$. We can use Lem. 4.5.2(1).

If the minimum local rule involves shallow edges that are from $\vec{H}^1$, endpoints of these edges are reachable from the token target. This means that, in the context $\dot{\mathcal{C}}$, some holes are shallow and their sources are reachable from the token target. Moreover, given that the token is not entering in $\dot{\mathcal{C}}$, the context has a path from the token target to a source of a hole edge.

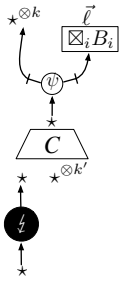



Figure 4.6: A shallow overlap ($C$ is a contraction tree, $B_i$ are box edges)

For example, in checking robustness of $\lhd^{\mathrm{BPerm}}$ with respect to copy transitions, one situation of shallow overlaps is when $\dot{G}$ is in the form of Fig. 4.6, and some of the box edges $B_i$ are from $\vec{H}^1$. Taking the minimum contraction rule means that $C$ in the graph is a contraction tree that gives a path from the token target. This path $C$ followed by the operation edge $\phi$ corresponds to paths from the token target to hole sources in the context $\dot{\mathcal{C}}$.

So, if the minimum local rule involves shallow edges that are from $\vec{H}^1$, the context $\dot{\mathcal{C}}$ necessarily has a path $P$ from the token target to a hole source. The path becomes a path in the state $\dot{\mathcal{C}}[\vec{H}^1]$, from the token target to a source of an edge $e$ that is from $\vec{H}^1$. The edge $e$ is necessarily shallow, and also involved in the application of the minimum local rule, because of the connectivity of $\dot{G}$. Moreover, a source of the edge

$e$ is an input, in the relevant hypernet of $\vec{H}^1$. By inspecting minimum local rules, we can enumerate possible labelling of the path $P$ and the edge $e$, as summarised in Table 4.4. Explanation on the notation used in the table is to follow.

| local rule | labels of path $P$ | label of edge $e$ |
|---|---|---|
| contraction | $(\otimes^\star_{\mathsf{C}})^+ \cdot \mathbb{O}^{\mathrm{ex}}$ | box |
| | $(\otimes^\star_{\mathsf{C}})^+$ | $\otimes^\star_{\mathsf{C}},\ \mathsf{I},\ \mathbb{O}^{\mathrm{ex}}$ |
| $\overrightarrow{@}$ | $\overrightarrow{@} \cdot \lambda$ | box |
| | $\overrightarrow{@} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^*$ | $\mathbb{O}^{\mathrm{ex}}_{\checkmark},\ \mathsf{I}$ |
| `ref` | $\mathtt{ref} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^*$ | $\mathbb{O}^{\mathrm{ex}}_{\checkmark},\ \mathsf{I}$ |
| $=$ | $=$ | $\mathsf{I}$ |
| | $= \cdot \mathsf{I} \cdot (\otimes^\diamond_{\mathsf{C}})^*$ | $\otimes^\diamond_{\mathsf{C}},\ \circ$ |
| $:=$ | $:= \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^* \cdot \mathsf{I} \cdot (\otimes^\diamond_{\mathsf{C}})^*$ | $\otimes^\diamond_{\mathsf{C}},\ \circ$ |
| | $:= \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^*$ | $\mathbb{O}^{\mathrm{ex}}_{\checkmark},\ \mathsf{I}$ |
| $!$ | $!$ | $\mathsf{I}$ |
| | $! \cdot \mathsf{I} \cdot (\otimes^\diamond_{\mathsf{C}})^*$ | $\otimes^\diamond_{\mathsf{C}},\ \circ$ |
| $+$ | $+$ | $\mathbb{Z}$ |
| $-$ | $-$ | $\mathbb{Z}$ |
| $-_1$ | $-_1$ | $\mathbb{Z}$ |

Table 4.4: Summary of paths that witness shallow overlaps

We use the regular-expression like notation in Table 4.4. For example, $(\otimes^\star_{\mathsf{C}})^+ \cdot \mathbb{O}^{\mathrm{ex}}$ represents finite sequences of edge labels, where more than one occurrences of the label $\otimes^\star_{\mathsf{C}}$ is followed by one operation $\phi \in \mathbb{O}^{\mathrm{ex}}$. This characterises paths that inhabit the overlap shown in Fig. 4.6, i.e. the contraction tree $C$ followed by the operation edge $\phi$. Note that this regular-expression like notation is not a proper regular expression, because it is over the infinite alphabet $M_{\mathbb{O}^{\mathrm{ex}}}$, the edge label set, and it accordingly admits infinite alternation (aka. union) implicitly.

To wrap up, checking robustness of each pre-template (that are not on contraction trees) boils down to using Lem. 4.5.2(1) and/or analysing the cases enumerated in Table 4.4.

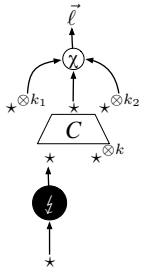



Figure 4.7: A focussed context ($C$ is a contraction tree)

**Robustness of $\lhd^{\otimes}$ and its converse**

Robustness check of the pre-template $\lhd^{\otimes}$ with respect to copy transitions has two cases. The first case is when one shallow overlap is caused by a path characterised by $(\otimes^{\star}_{\mathsf{C}})^{+}$, and the second case is when no shallow overlaps are present and Lem. 4.5.2(1) can be used.

In the first case, namely, a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen with a non-entering rewrite token can be turned into the form $(\mathcal{C}[\dot{\mathcal{C}}'[\chi'], \vec{\chi}]; H^1, \vec{H}^1; H^2, \vec{H}^2)$ where $j$ is a positive number, and $\dot{\mathcal{C}}'$ is a focussed context in the form of Fig. 4.7. A rewrite transition is possible on both states given by the specimen, in which a contraction rule is applied to $\dot{\mathcal{C}}'[H^1]$ and $\dot{\mathcal{C}}'[H^2]$. Results of the rewrite transition give a new quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen. When $k_1 = 0$, this quasi-specimen is up to $(=, \dot{\simeq}_{=_{\mathbb{N}}})$, using the trigger $\lhd^2$. When $k_1 > 0$, the quasi-specimen is also up to $(=, \dot{\simeq}_{=_{\mathbb{N}}})$, but using the trigger $\lhd^1$.

Robustness check of the pre-template $\lhd^{\otimes}$ with respect to rewrite transitions always boils down to Lem. 4.5.2(1). This is intuitively because no local rewrite rule of operations involves any shallow contraction edge of type $\star$.

Robustness of $(\lhd^{\otimes})^{-1}$ can be checked in a similar manner. Namely, using Table 4.4, shallow overlaps are caused by paths:

| | | |
|---|---|---|
| $(\otimes^{\star}_{\mathsf{C}})^{+}$, | $=$, | $+$, |
| $\overrightarrow{@} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$, | $:= \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$, | $-$, |
| $\mathtt{ref} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$, | $!$, | $-_1$ |

from the token target. All paths but $(\otimes^\star_{\mathsf{C}})^+$ give rise to a state that is not rooted, which can be checked using Lem. 4.3.17. This reduces the robustness check of $(\lhd^{\otimes})^{-1}$ to that of $\lhd^{\otimes}$.

**Robustness of $\lhd^{\mathrm{GC}}$ and $\lhd^{\mathrm{NE2}}$, and their converse**

These four pre-templates both relate hypernets with no inputs. Proofs of robustness of them and their converse always boils down to the use of Lem. 4.5.2(1), following the discussion in Sec. 4.5.5. Namely, it is impossible to find the path $P$ in the context $\dot{\mathcal{C}}$ from the token target to a hole source.

**Robustness of $\lhd^{\mathrm{BPerm}}$, $\lhd^{\mathrm{BPullC}}$, $\lhd^{\mathrm{BPullW}}$ and $\lhd^{\mathrm{BPullD}}$, and their converse**

These eight pre-templates all concern boxes. Using Table 4.4, shallow overlaps are caused by paths $(\otimes^\star_{\mathsf{C}})^+ \cdot \mathbb{O}^{\mathrm{ex}}$ and $\overrightarrow{@} \cdot \lambda$ from the token target.

Robustness check with respect to compute transitions of operations $\mathbb{O}^{\mathrm{ex}}_{\not\iota} \backslash \{\overrightarrow{@}\}$ always boil down to Lem. 4.5.2(1).

As for compute transitions of the operation '$\overrightarrow{@}$', either one path $\overrightarrow{@} \cdot \lambda$ causes one shallow overlap, or all overlaps are deep. The latter situation boils down to Lem. 4.5.2(1). In the former situation, a beta rule involves one box that is contributed by a pre-template, and states given by a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen are turned into a quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=,=)$, by one rewrite transition.

As for copy transitions, there are two possible situations.

- Paths $(\otimes^\star_{\mathsf{C}})^+ \cdot \mathbb{O}^{\mathrm{ex}}$ cause some shallow overlaps and there are some deep overlaps too.
- All overlaps are deep, which boils down to Lem. 4.5.2(1).

In the first situation, some of the shallow boxes duplicated by a contraction rule are contributed by a pre-template, and other duplicated boxes may have deep edges contributed by the pre-template. By tracking these shallow and deep contributions

in a contraction rule, it can be checked that one rewrite transition turns states given by a $\mathbb{C}_{\mathbb{O}^{\text{ex}}}$-specimen into a quasi-$\mathbb{C}_{\mathbb{O}^{\text{ex}}}$-specimen. This quasi-specimen is up to the following, depending on pre-templates:

- $(=, =)$ for $\lhd^{\text{BPerm}}$ and its converse,
- $(=, \dot{\simeq}_{=_{\mathbb{N}}})$ for $\lhd^{\text{BPullC}}$, and $(\dot{\simeq}_{=_{\mathbb{N}}}, =)$ for its converse, using the trigger $\lhd^3$,
- $(=, \dot{\simeq}_{=_{\mathbb{N}}})$ for $\lhd^{\text{BPullW}}$, and $(\dot{\simeq}_{=_{\mathbb{N}}}, =)$ for its converse, using the trigger $\lhd^4$, and
- $(=, \dot{\preceq}_{\leq_{\mathbb{N}}})$ for $\lhd^{\text{BPullD}}$, and $(\dot{\preceq}_{\geq_{\mathbb{N}}}, =, )$ for its converse, using the trigger $\lhd^5$.

**Robustness of operational pre-templates and their converse**

For the operational pre-templates and their converse, we use the class $\mathbb{C}_{\mathbb{O}^{\text{ex}}\text{-bf}}$ of binding-free contexts. This restriction is crucial to rule out some shallow overlaps.

Using Table 4.4, shallow overlaps with the operational pre-templates $\lhd^{\vec{@}}$ and $\lhd^{\mathtt{ref}}$ are caused by paths $(\otimes^\star_{\mathsf{C}})^+$ from the token context. However, the restriction to binding-free contexts makes this situation impossible, which means the robustness check always boils down to Lem. 4.5.2(1) and Lem. 4.5.2(2).

In checking robustness of the converse $(\lhd^{\vec{@}})^{-1}$ and $(\lhd^{\mathtt{ref}})^{-1}$, shallow overlaps are caused by paths:

| | | |
|---|---|---|
| $(\otimes^\star_{\mathsf{C}})^+$, | $=$, | $+$, |
| $\vec{@} \cdot (\mathbb{O}^{\text{ex}}_{\checkmark})^*$, | $:= \cdot (\mathbb{O}^{\text{ex}}_{\checkmark})^*$, | $-$, |
| $\mathtt{ref} \cdot (\mathbb{O}^{\text{ex}}_{\checkmark})^*$, | !, | $-_1$ |

from the token target. Like the case of $(\lhd^{\otimes})^{-1}$, all paths but $(\otimes^\star_{\mathsf{C}})^+$ give rise to a state that is not rooted, which can be checked using Lem. 4.3.17. The paths $(\otimes^\star_{\mathsf{C}})^+$ are impossible because of the binding-free restriction. As a result, this robustness check also boils down to Lem. 4.5.2(1) and Lem. 4.5.2(2).

**Robustness of name-exhaustive pre-templates on lambda-abstractions, and their converse**

We here look at the three name-exhaustive pre-templates $\lhd^{\mathrm{NE1}}$, $\lhd^{\mathrm{NE3}}$ and $\lhd^{\mathrm{NE4}}$, and their converse. All these six pre-templates concern lambda-abstractions and their different use of a name. The pre-templates $\lhd^{\mathrm{NE1}}$, $\lhd^{\mathrm{NE4}}$ and their converse compare lambda-abstractions that all refer to a single name, with lambda-abstractions that instantly return a value without using the name at all. The pre-template $\lhd^{\mathrm{NE3}}$ and its converse compare lambda-abstractions that increment a number associated with a name, with lambda-abstractions that decrement the number. The latter lambda-abstractions additionally flip the sign of the number after dereferencing the name. As a result, all the six pre-templates give rather rare examples of robustness check where we compare different numbers of computation steps, i.e. transitions.

Using Table 4.4, shallow overlaps with these pre-templates are caused by paths:

$$(\otimes^{\star}_{\mathsf{C}})^{+}, \qquad \vec{@} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*},$$
$$\mathtt{ref} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}, \qquad := \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$$

from the token target.

As for compute transitions of operations $\mathbb{O}^{\mathrm{ex}}_{\lightning}\backslash\{\vec{@}\}$, there are two possible situations.

- Shallow overlaps are caused by paths $\mathtt{ref} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$ or $:= \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^{*}$.
- There is no overlap at all, which boils down to Lem. 4.5.2(1).

In the first situation, a stable hypernet $G_S$ of a local rewrite rule (see e.g. Fig. 3.9) contains shallow edges, labelled with $\lambda \in \mathbb{O}^{\mathrm{ex}}_{\checkmark}$, that are contributed by a pre-template. The overlapped shallow contributions are not modified at all by the rewrite rule, and consequently, one rewrite transition results in a quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=, =)$.

As for copy transitions, either one path $(\otimes^\star_{\mathsf{C}})^+$ causes one shallow overlap, or all overlaps are deep. The latter situation boils down to Lem. 4.5.2(1). In the former situation, one lambda-abstraction contributed by a pre-template gets duplicated. Namely, a $\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen with a non-entering rewrite token can be turned into the form $(\mathcal{C}[\dot{\mathcal{C}}'[\chi'], \vec{\chi}]; H^1, \vec{H}^1; H^2, \vec{H}^2)$ where $\dot{\mathcal{C}}'$ is a focussed context in the form of Fig. 4.7. There exist a focussed context $\dot{\mathcal{C}}''$ and two hypernets $G^1 \lhd^{\mathrm{NE1}} G^2$ such that:

$$\mathcal{C}[\dot{\mathcal{C}}'[H^1], \vec{H}^1] \to \mathcal{C}[\dot{\mathcal{C}}''[G^1], \vec{H}^1],$$
$$\mathcal{C}[\dot{\mathcal{C}}'[H^2], \vec{H}^2] \to \mathcal{C}[\dot{\mathcal{C}}''[G^2], \vec{H}^2].$$

Results of these rewrite transitions give a new quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=, =)$.

As for compute transitions of the operation '$\vec{@}$', there are two possible situations.

- One path $\vec{@}$ causes a shallow overlap of the edge that has label $\lambda$ and gets eliminated by a beta rewrite rule, and possibly some other paths $\vec{@} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^*$ cause shallow overlaps in the stable hypernet $G_S$ (see Fig. 3.7).

- There are possibly deep overlaps, and paths $\vec{@} \cdot (\mathbb{O}^{\mathrm{ex}}_{\checkmark})^*$ may cause shallow overlaps in the stable hypernet $G_S$.

In the second situation, all overlaps are not modified at all by the beta rewrite rule, except for some deep overlaps turned shallow. Consequently, one rewrite transition results in a quasi-$\mathbb{C}_{\mathbb{O}^{\mathrm{ex}}}$-specimen up to $(=, =)$.

In the first situation, one lambda-abstraction contributed by the pre-template is modified, while all the other shallow overlaps (if any) are not. We can focus on the lambda-abstraction. The beta rewrite acts on the lambda-abstraction, an edge labelled with '$\vec{@}$', and the stable hypernet $G_S$.

In the case of $\lhd^{\mathrm{NE1}}$, application of the beta rule is followed by a few more transitions. When $G_S$ is not an instance edge, we can prove that a token eventually

gets stuck, failing to apply a rewrite rule of the equality operation '='. When $G_S$ is simply an instance edge, a token may still get stuck for the same reason, but if there is an applicable rewrite rule of the equality operation '=', we obtain a quasi-$\mathbb{C}_{\mathbb{O}^{\text{ex}}}$-specimen up to $(\dot{\simeq}_{=_{\mathbb{N}}}, \dot{\simeq}_{=_{\mathbb{N}}})$, using triggers $\lhd^6$ and $\lhd^7$ on the left, and $\lhd^{\text{GC}}$ on the right.

The case of $\lhd^{\text{NE3}}$ and $\lhd^{\text{NE4}}$, and their converse, is similar. The only difference is the triggers used to find a quasi-specimen, as summarised in Table 4.3.

# Chapter 5

# Discussion

## 5.1 Linear-logic concepts, from the DGoIM to the UAM

While the DGoIM is proposed as a framework for execution cost analysis, the concept of locality it offers leads to the development of the UAM, the machine for direct equational reasoning. Concepts taken from linear logic [Girard, 1987], namely the !-box structure and generalised contractions, are essential for the rewrites-first DGoIM to achieve time efficiency. Based on the rewrites-first DGoIM, the UAM also benefits from these concepts, but not necessarily in the same way.

### 5.1.1 Box structures

The DGoIM uses a box structure to represent lambda-abstractions, which are the only values of the pure lambda-calculus, via the so-called *call-by-value translation* of linear logic to intuitionistic logic (e.g. [Maraist et al., 1999]). In the lambda-calculus with the call-by-need or call-by-value evaluation, which the DGoIM aims at in Chap. 2, lambda-abstractions serve as the unit of duplication and delay; each function body gets copied as a whole, and never gets evaluated until an argument is provided by function application. The boxes hence indicate values, computation

to be duplicated, and computation to be delayed.

The UAM employs a box structure for a restricted purpose. Boxes used by the UAM correspond not to values but thunks, which are identified as the unit of delay in the language SPARTAN. Thunks do not themselves become a unit of duplication, but they provide enough information to determine which sub-graphs to duplicate. The UAM does not rely on boxes to determine and trace values during execution. Values are instead characterised by passive operations, which never trigger any rewrite.

### 5.1.2 Generalised contractions as optimisation

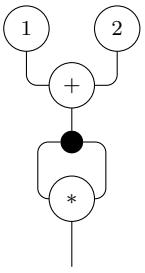
Time efficiency of the rewrites-first DGoIM crucially relies on the translation of terms to graphs that introduces exactly one generalised contraction per variable (see Fig. 2.11). In comparison, the UAM takes a more primitive approach. Multiple occurrences of a single variable in SPARTAN is represented by a combination of binary contractions and weakenings as in proof nets [Girard, 1987].

This deviation is because of the different purposes these machines have. The UAM together with the core language SPARTAN provides a rather primitive framework, in which observational equivalence can be proved and sophistication of language features can be justified using the equivalence. In a language as flexible as SPARTAN, observational equivalences are not guaranteed to hold in any settings.

The rewrites-first DGoIM is implicitly optimised to achieve time efficiency, whose safety could be justified by the UAM in terms of observational equivalence. Indeed, equivalences implied by some templates in Sec. 4.5.2 seem to be helpful in justifying the optimisation. The equivalences implied by templates $\lhd^{\text{Assoc}}$, $\lhd^{\text{Comm}}$ and $\lhd^{\text{Idem}}$ enable us to identify certain contraction trees of the same number of inputs, and hence to think of one generalised contraction to denote these all. Additionally, the equivalences implied by templates $\lhd^{\text{BPullC}}$ and $\lhd^{\text{BPullW}}$ would enable us to merge generalised contractions across the boundary of the box structure, and hence to introduce only one generalised contraction to represent a single variable.

### 5.1.3 Interaction with contractions

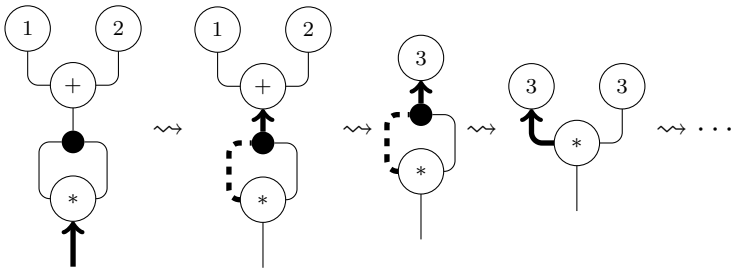
Contractions, whether generalised or not, are used to represent multiple occurrences of a variable, and to graphically connect the occurrences to computation associated with the variable. For example, the following graph represents a term $x * x$ whose variable $x$ is associated with computation $1 + 2$, using an idealised contraction $\bullet$:




From the perspective of the DGoIM, the idealised contraction $\bullet$ denotes a generalised contraction, and the graph represents a term $(x*x)[x \leftarrow 1+2]$. From the perspective of the UAM, the idealised contraction $\bullet$ denotes a contraction tree, and the graph represents a term $\texttt{bind}\ x \to 1 + 2\ \texttt{in}\ x * x$.

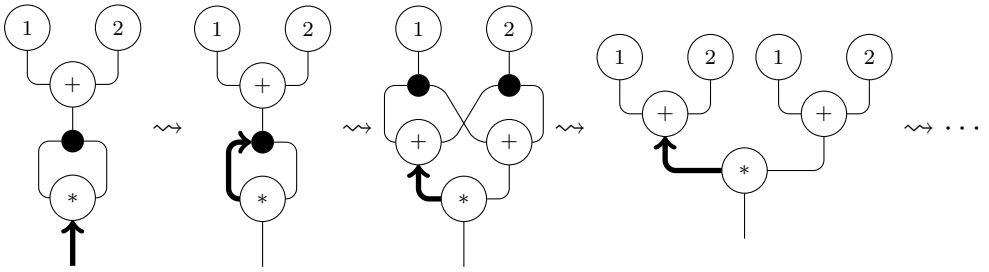
In strategical graph rewriting, there is a choice of the way the token interacts with contractions. This affects evaluation of computation associated with a variable, in other words, computation shared via a variable. The two abstract machines DGoIM and UAM, in fact, support different interactions between the token and contractions. In the following, the difference is informally illustrated using the above example graph. The token position is depicted as a thick arrow, the token data is mostly omitted, and possibly multiple transitions are represented by $\rightsquigarrow$.

In the rewrites-first DGoIM, the token passes through the contraction $\bullet$, visits and reduces the sub-graph representing $1 + 2$, and then duplicates the result:



When the token passes through the contraction •, it remembers its old position, as indicated by a dashed line. This record is necessary to determine a token position after the duplication. In the rewrites-first DGoIM, the record of previous positions is the only part of the token data that depends on underlying graphs, namely a box stack (see Def. 2.3.1).

On the other hand, in the UAM, the token never passes through the contraction •, which is a contraction tree, but instead immediately triggers duplication. The duplication starts with '+', and as the token continues travelling, the whole sub-graph representing $1 + 2$ is eventually duplicated:




The token then proceeds to reduce each copy of $1 + 2$ separately, which means repeated evaluation of the computation $1 + 2$ that was originally shared. Despite the inefficiency of repeated evaluation, the immediate trigger of duplication leads to simpler token data. The token does not need to record any previous position, and as a result, the token data of the UAM is completely independent from underlying graphs.

To summarise, the rewrites-first DGoIM takes a *reduction-oriented* approach, the UAM takes a *duplication-oriented* approach, and these approaches are governed by the way the token interacts with contractions.

The reduction-oriented approach avoids repeated evaluation, which is essential for the rewrites-first DGoIM to implement the call-by-need evaluation efficiently. On the other hand, the UAM benefits from simpler token data, which is made possible by the duplication-oriented approach. The token data of the UAM is as simple as an edge label (?, ✓ or $\lightning$), which is independent from underlying graphs. This independence greatly simplifies the local, case-by-case, inspection of token moves

and triggered rewrites, which is at the core of observational equivalence proofs.

Recall that the duplication-oriented approach is at the cost of repeated evaluation. This results in the call-by-name binding, instead of the call-by-need binding, native to SPARTAN and the UAM. Unlike the call-by-need binding, the call-by-name binding may enable contexts to distinguish computation from values. The restriction of contexts to binding-free contexts (Def. 4.2.3, with syntactical explanation in Sec. 4.2.1) provides a way to avoid this extra distinguishing power from invalidating some observational equivalences, such as the call-by-value beta law.

## 5.2 Case study: data-flow networks

The DGoIM is introduced as a framework that combines token passing with graph rewriting in a flexible and disciplined way. It turns out that this combination can naturally model unconventional but increasingly significant paradigms of programming, namely programming with *data-flow networks*. In addition to conventional computation, such a style of programming involves construction, manipulation and observation of data-flow networks. This section briefly explains a case study on accommodating the data-flow networks in the context of token-guided graph rewriting [Cheung et al., 2018, Muroya et al., 2018].

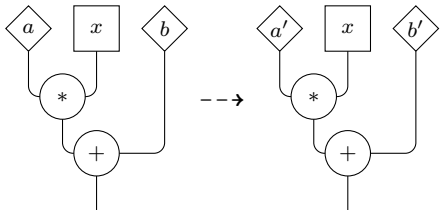
### 5.2.1 TensorFlow networks

TENSORFLOW, a machine-learning library developed by Google[1], is a successful example in which machine-learning models are managed as parametrised data-flow networks. A constructed model has input, output, and additionally, parameters. It can be used in two ways: for *training*, where parameters get updated so that the model better describe given knowledge about input and output; and for *prediction*, where the output is computed with respect to the current parameters and given

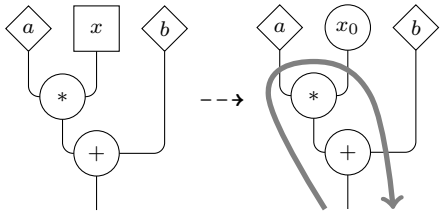
[1] `https://www.tensorflow.org/`

input. This dual usage of the model can be understood in terms of manipulation and observation of the corresponding network.

Informally, a linear regression model $f(x) = a * x + b$ with two parameters $a$ and $b$ can be represented as a network on the left below, where the input $x$ is denoted by a rectangle and the parameters are denoted by diamonds. Computation on these elements is graphically described with the two circle nodes that denote operations $*$ and $+$. Training this model results in updating the two parameters $a$ and $b$ (to, say, $a'$ and $b'$), and this update can be seen as the following simple manipulation of the network:




Given parameters $a$ and $b$, and actual input data $x_0$, predicting output amounts to observation following subtle manipulation. The subtle manipulation is the replacement of the rectangle node that denotes the input with a circle node that denotes the actual value $x_0$, as depicted below. After this, the output data $f(x_0) = a * x_0 + b$ can be read back from the network on the right, by an in-order traversal of the graph as indicated by a thick grey arrow:




TensorFlow, as an embedded domain specific language, provides a syntactical interface to construct and use the data-flow networks. The key idea that underlies these networks is the classification of nodes into three, as described above: computation nodes (circles) that denote operations and constant values, which can be multi-dimensional arrays; input nodes (rectangles) that are to be replaced with values; and parameter nodes (diamonds) that can be updated in place and also observed

as values. In the TensorFlow terminology, input nodes are referred to as *placeholders* and parameter nodes as *variables*. The following code, written in a simplified form of the Python binding of TensorFlow, describes the linear regression model $f(x) = a * x + b$ that is constructed with initial parameters $a = 1$ and $b = 0$, used once for prediction, trained, and used again for prediction with given input $x = 10$:

```
1  import tensorflow as tf
2  # construct the model
3  x = tf.placeholder(tf.float32)  # input 'x'
4  a = tf.Variable(1)              # parameter 'a'
5  b = tf.Variable(0)              # parameter 'b'
6  y = a * x + b
7  with tf.Session() as s:
8    # initialise parameters, using 'init' defined elsewhere
9    s.run(init)
10   # train the model, using 'train' defined elsewhere
11   s.run(train)
12   # predict output with the updated model
13   y_0 = s.run(y, feed_dict={x: 10})
```

Parameter nodes are updated in-place in line 11, whose result is used in line 13 for prediction. Also in line 13, an input value 10 is associated with the input node 'x' using 'feed_dict'. Note that all these manipulation and observation are done single-handedly by calling 'run' within what is called *session*.

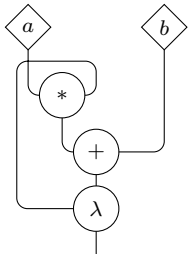
### 5.2.2 Parametrised networks in the DGoIM style

The construction, manipulation and observation of parametrised data-flow networks can be understood as combination of token passing and graph rewriting, from the perspective of the DGoIM that models the lambda-calculus. In collaboration with Steven W. T. Cheung, Victor Darvariu, Dan R. Ghica and Reuben N. S. Rowe, we formalise this as a token-guided graph-rewriting abstract machine *à la* DGoIM, which models an extension of the simply-typed lambda calculus Cheung et al. [2018], Muroya et al. [2018].

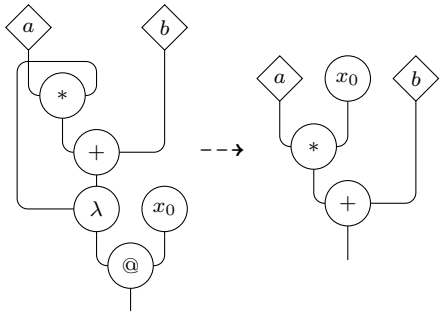
The extended calculus is dubbed *Idealised TensorFlow* (ITF). It has two novel language features, namely parameters and *graph abstraction*, to express the computa-

tion with parametrised data-flow networks. Its semantics, a variation of the DGoIM, accordingly has extra nodes that represent parameters, and an extra rewriting rule of graph abstraction. These extra features altogether model the behaviour of the parameter nodes in TensorFlow network, in a functional way. The rest of this section gives an informal description using a simplified style of the DGoIM graphs, deliberately ignoring their box structure, and making the token implicit in rewriting rules.

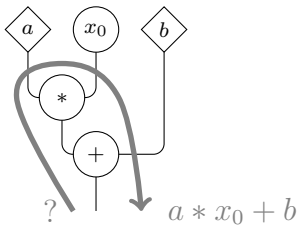
The starting point is to get rid of one of the three classes of nodes in the TensorFlow network, namely input nodes. They can be replaced with the nodes for lambda-abstraction and function application *à la* DGoIM. The linear regression model $f(x) = a * x + b$ can be represented as a lambda-abstraction with two parameter nodes:




The subtle manipulation required by prediction, which was to replace the input node with a value node $x_0$, can be simply modelled by graphical beta reduction:

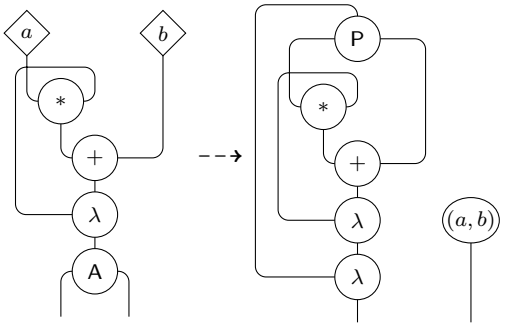



Observation of the resulting network, which involves parameter nodes, can be achieved solely by token passing. The output data $a * x_0 + b$ can be obtained by letting the token travel through the network from the bottom, as indicated by a thick gray arrow:



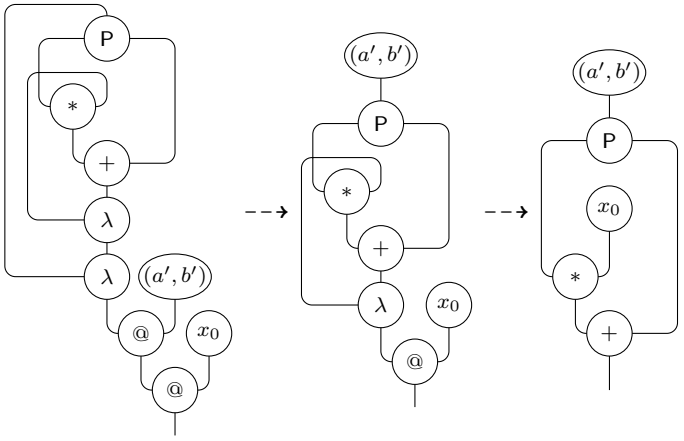

Values in gray ('?' and '$a * x_0 + b$') represents token data, enriched to record values; recall that the token uses its data to determine routing and control rewriting in the DGoIM. The token can itself read back a value from the network, if it can record a value of a node and perform an operation denoted by a node as it travels through the network.

The main manipulation of parametrised data-flow networks is to update parameter nodes. This can be modelled as a combination of graphical beta reduction and *graph abstraction*, a new graph-rewriting rule. Graph abstraction "abstracts away" all the parameters of a network, and turns the parametrised network into an ordinary network that represents a function on vectors. As a side product, it creates a value node that represents a vector whose elements are the current values of the parameters. The rewriting rule is formalised using two additional nodes: the node 'A' dedicated to trigger the rewrite, and a node 'P' that denotes projections of a vector. When applied to the linear regression model, the rule looks like below:




Graph abstraction is not a local rewriting rule, because it extracts *all* parameters in a network, which are not necessarily neighbours of the triggering node 'A'. Parameters are extracted altogether as a function argument, so that parameter update can be completed with graphical beta reduction. This deviates from the in-place update of TensorFlow. Once a new parameter vector $(a', b')$ is computed, pre-

diction with the updated model $f(x) = a' * x + b'$ using input data $x_0$ can be started with two steps of graphical beta reduction:




This will be followed by projections of the new parameter vector into each element.

The calculus ITF is proposed as an extension of the simply-typed lambda-calculus, in which the computation described above can be expressed with two extra language features: parameters and graph abstraction. The TensorFlow code in Sec. 5.2.1 corresponds to the following program in ITF, using the OCaml-like convention:

```
1  ;; construct the model as a function with two parameters
2  let a = {1} in
3  let b = {0} in
4  let y = fun x -> a * x + b in
5  ;; turn the model into a function and a parameter vector
6  let (model,p) = abs y in
7  ;; update the parameter vector with 'train' defined elsewhere
8  let q = train model p in
9  ;; predict output with the updated model
10 let y_0 = model q 10 in
11 y_0
```

The parameters, indicated by $\{-\}$, enable users to construct a model without explicitly declaring which parameters are involved in each model. This convenient style of construction is taken from TensorFlow. What used to be in-place update of parameters in TensorFlow is now decomposed through graph abstraction, which can be accessed by programmers using the operation '`abs`' (line 6).

As a final remark, the idea of local reasoning, which is extensively investigated in Chap. 4, was initially tried out with ITF and its DGoIM-style model. Although

| ITF term using parameters | SPARTAN term using names |
|---|---|
| `let` a `=` $\{1\}$ `in` `(fun` x `->` a $*$ x $+$ a`)` | `new` $a \multimap 1$ `in` $\lambda x.\, !a \times x + !a$ |
| `fun` x `->` $\{1\} * \mathrm{x} + \{0\}$ | `new` $a \multimap 1$ `in new` $b \multimap 0$ `in` $\lambda x.\, !a \times x + !b$ |
| | $\lambda x.$ (`new` $a \multimap 1$ `in new` $b \multimap 0$ `in` $!a \times x + !b$) |

Table 5.1: Parameters, and their possible representation with name binding

the model is a variation of the DGoIM, it is not as tuned for efficiency as the DGoIM in the way discussed in Sec. 5.1.2. The model is rather used to prove soundness of ITF programs (recall that ITF is simply-typed) and some observational equivalence, namely garbage collection and a restricted form of the beta law. The proof technique introduced for observational equivalence on ITF programs is based around the concept of local reasoning, which inspired the development of the UAM and the characterisation theorem (Thm. 4.3.14).

It would be interesting to reformulate ITF and its semantics using SPARTAN and the UAM, which seems possible but not straightforward. Graph abstraction could be modelled as an extrinsic operation of SPARTAN that has global behaviour, i.e. a behaviour that cannot be specified by a local rewrite rule. It is tempting to represent parameters of ITF in SPARTAN using name binding and the extrinsic operation for dereferencing. However, it seems that the representation should be a non-trivial, global, one. Table 5.1 shows some illustrating examples.

The first ITF term in Table 5.1 represents a model with one parameter that is named 'a' and used twice. The multiple occurrences of the name do not mean duplication of the parameter '{1}' itself, which matches the sharing behaviour of name binding in SPARTAN. The ITF term therefore seems to correspond to the similar SPARTAN term (namely, the first SPARTAN term in Table 5.1) that introduces the bound name $a$ and dereferences it twice.

However, parameters can also be introduced and used anonymously in ITF, like the second ITF term in Table 5.1. Anonymous parameters could be represented in SPARTAN by introducing fresh name bindings, but there is not a single way to

do so. The table shows two possible ways: introducing name bindings outside the lambda-abstraction, and inside the lambda-abstraction. These two representations have different behaviours in SPARTAN, because the sharing behaviour of name binding varies according to where the name binding is placed, as explained in Sec. 3.2.2. It is in fact the first representation, which places fresh name bindings all outside the lambda-abstraction, that achieves the same behaviour as the original ITF term. This suggests that it would require some global perspective to appropriately introduce name bindings, so that ITF parameters, especially anonymous parameters, are properly represented in SPARTAN.

# Chapter 6

# Related and future work

## 6.1 Environments in abstract machines

In an abstract machine of any functional programming language, computations assigned to variables have to be stored for later use. This storage, often called *environment*, is expanded when the abstract machine encounters a new variable, and referred to when the machine encounters a known variable. The environment needs to be carefully managed throughout program execution, so that each variable is associated with unique computation in the environment, otherwise there would be conflicting results of looking up a variable.

However, naive management of the environment would generate such conflicting entities. For example, executing a functional program $(\lambda f.\,(f\,0)+(f\,1))\,(\lambda x.\,x)$ would apply the identity function $\lambda x.\,x$ twice with different arguments 0 and 1. This means that, naively, both these arguments would be associated with the same variable $x$ in the environment.

Different solutions to this conflict lead to different representations of the environment, some of which are examined by Accattoli and Barras [2017] from the perspective of time-cost analysis. A few solutions seem relevant to token-guided graph rewriting.

One solution is to allow at most one assignment to each variable. This is typi-

cally achieved by renaming bound variables during execution, possibly symbolically. Examples for call-by-need evaluation are Sestoft's abstract machines [Sestoft, 1997], and the storeless and store-based abstract machines studied by Danvy and Zerny [2013]. The graph-rewriting abstract machines presented in this thesis give another example. This is shown, in the case of the rewrites-first DGoIM, by the simulation of the sub-machine semantics that resembles the storeless abstract machine mentioned above. Variable renaming is trivial in both the DGoIM and the UAM, thanks to the use of graphs, in which variables are represented anonymously by mere edges.

Another solution is to allow multiple assignments to a variable, with restricted visibility. The common approach is to pair a sub-term with its own localised environment that maps its free variables to their assigned computations, forming a so-called *closure*. Conflicting assignments are distributed to distinct localised environments. Examples include Cregut's lazy variant [Crégut, 2007] of Krivine's abstract machine for call-by-need evaluation, and the SECD machine of Landin [1964] for call-by-value evaluation. Fernández and Siafakas [2009] refine this approach for call-by-name and call-by-value evaluations, based on closed reduction [Fernández et al., 2005], which restricts beta-reduction to closed function arguments. This suggests that the approach with localised environments can be modelled with token-guided graph rewriting by implementing closed reduction. The implementation would require the ability to manipulate boxes, especially to merge them.

Finally, Fernández and Siafakas [2009] propose another approach to multiple assignments, in which multiple assignments are augmented with binary strings so that each occurrence of a variable can only refer to one of them. This approach is inspired by the token-passing GoI, namely a token-passing abstract machine for call-by-value evaluation, designed by Fernández and Mackie [2002]. The augmenting binary strings come from paths of trees of binary contractions, which are used by the token-passing machine, as well as the UAM, to represent shared assignments.

## 6.2 Graph rewriting with boxes and token

The box structures used by the DGoIM and the UAM are inspired by the *exponential boxes* of proof nets, a graphical representation of linear logic proofs [Girard, 1987]. In the framework of proof nets, and an established graph-rewriting framework of *interaction nets* [Lafont, 1990] that subsume proof nets, several graphical representations of exponential boxes have been proposed. Lafont [1995] formalises boxes by parametrising an agent (which corresponds to an edge of hypernets) by another net, and Mackie [1998] introduces coordinated agents that altogether represent a boundary of a box. Accattoli and Guerrini [2009] proposes a box structure that is represented by extra edges. Each of these approaches is relevant to the DGoIM or the UAM.

In the rewrites-first DGoIM, boxes are formalised by coordinating nodes labelled with '!' and '?', which resembles Mackie's approach. However, cost analysis of the DGoIM in Sec. 2.5 adopts the view of boxes as extra edges to achieve efficiency, sharing the idea with the approach of Accattoli and Guerrini.

Boxes of the UAM, on the other hand, are closely related to Lafont's exponential boxes. In comparison with exponential boxes, the boxes of hypernets, namely box edges, have flexibility regarding types of a box edge itself and its content (i.e. the hypernet that labels it). Each box edge represents a thunk, and it can have less targets than outputs of its contents, reflecting the number of bound variables the thunk has.

The idea of using the token as a guide of graph rewriting was also proposed by Sinot [2005, 2006] for interaction nets. He shows how using a token can make the interaction-net rewriting system implement the call-by-name, call-by-need and call-by-value evaluation strategies. The rewrites-first DGoIM can be seen as a realisation of the rewriting system as an abstract machine, in particular with explicit control over copying sub-graphs. As a revision of the rewrites-first DGoIM, the UAM could

possibly be formalised with interaction nets. However, local reasoning does not seem as easy in interaction-net rewriting as in the UAM, because of technical subtleties observed in *loc. cit.* Namely, a status of evaluation is remembered by not only the token but also some other agents around an interaction net, which blurs locality of information.

A similar structure to the box structure of the UAM is studied by Drewes et al. [2002] as *hierarchical graphs*, in the context of double-pushout graph transformation [Rozenberg, 1997], a well-established algebraic approach to graph rewriting. Investigating local reasoning in this context is an important future direction.

The double-pushout approach has been used to rewrite *string diagrams* [Kissinger, 2012, Bonchi et al., 2016], which provide graphical representation that can accommodate some *built-in* equations. The graphs used by the DGoIM and the UAM could also perhaps be formalised as string diagrams, with boxes modelled as *functorial boxes* [Melliès, 2006]. Nevertheless, local reasoning rather aims at *discovering* such built-in equations on graphs that represent programs, because it is not clear what should be, or can be, such a built-in equation in the presence of arbitrary language features.

## 6.3 Extrinsic operations

Extrinsic operations of Spartan are greatly inspired by *algebraic operations* [Plotkin and Power, 2001, 2003]. Algebraic operations are introduced as a syntactic interface to computational effects, such as non-determinism, I/O, exception and state. In the most general form, algebraic operations do have eager arguments, as well as deferred arguments with bound variables. They use eager arguments to determine effectful behaviour, and then continue computation with deferred arguments.

Algebraic operations provide a view of language features, namely effects, as *behaviour* of operations. This is in contrast to the view of language features as encoding

into the host language, that is to say, features of a language are described within the language. In the case of computational effects, this means encoding effects into a pure language, which can be achieved via monads [Moggi, 1988].

Spartan takes the behavioural view to the extreme, to the level that only binding of variables and names, and thunking, are intrinsic. Everything else becomes extrinsic operations, which have the same form as algebraic operations, and extrinsic operations are specified in terms of behaviour. The behaviour is represented as extrinsic transitions of the UAM, which are focussed graph-rewriting rule. The UAM benefits its "universality" from this extreme behavioural view of language features. It can model language features in a uniform way, whether they are effectful or pure, whether they are encoded or native.

## 6.4 Observations of program execution

Although the UAM itself can accommodate computational effects by means of extrinsic operations, its local reasoning principle was formalised in Chap. 4 for only the UAM that is deterministic. This restriction was primarily to keep the technical development relatively simple, in particular the notion of contextual refinement presented in Sec. 4.2 and step-wise reasoning presented in Sec. 4.4.

It is important future work to broaden the scope of local reasoning, by lifting the current restriction to sequential and deterministic computation. This would require an expanded model of program execution, and more significantly, a new definition of observational equivalence. The principle of local reasoning is expected to be still valid, but its details, such as the variant of simulation (Def. 4.4.1), would require a minor adaptation.

Parallelism and concurrency can be modelled with multiple tokens, which are travelling around a graph at the same time, as shown by Dal Lago et al. [2017] in the case of token-passing GoI. Non-deterministic computation, or computation

with probability or I/O, would require a minor extension of the UAM to be a non-deterministic and labelled transition system.

A significant modification required by these computations would rather be on a definition of observational equivalence, because these computations enrich observation of program executions with probability, input/output sequences, etc. Such a definition is studied by Johann et al. [2010] for algebraic effects, which include these computations.

## 6.5 Time and space efficiency

Cost analysis of the rewrites-first DGoIM, carried out in Chap. 2, primarily focussed on time efficiency. This is to complement existing work on operational semantics given by token-passing GoI, which usually achieves space efficiency, and also to confirm that introduction of graph rewriting to token passing does not bring in any hidden inefficiencies.

Only the rewrites-first and passes-only interleaving of graph rewriting with token passing have been studied, and flexible interleaving is yet to be explored. For instance, the DGoIM could choose between token passing and graph rewriting at each step of execution, taking its resource usage into account. One possible interleaving strategy would be to choose graph rewriting, in particular duplication of sub-graphs, as long as there is enough space left. It is future work to study the DGoIM with flexible interleaving, as a model of program execution under various time and space constraints.

Moreover, interleaving is not the only source of flexibility for the DGoIM. Each of its components, token passing and graph rewriting, could be adapted to serve particular objectives in the trade-off between time and space efficiency. As discussed in Sec. 5.1.3, different approaches to token passing, with respect to contractions, lead to different evaluation strategies regarding variable binding. Accommodation of call-

by-need variable binding in the UAM and SPARTAN is an interesting topic. Graph rewriting could also be refined, in terms of management of boxes for instance, to serve further objectives such as: *full lazy evaluation*, whose implementation with interaction nets and the token is studied by Sinot [2005]; and *optimal reduction* [Lévy, 1980, Lamping, 1990], whose relation to GoI is studied by Gonthier et al. [1992].

## 6.6 Improvement and optimisation

Although the UAM was not designed to study cost of program execution, one can think of a cost model of the machine in a similar way as the DGoIM. Additionally, the indexing of observational equivalence with a preorder that represents the number of execution steps paves the way for comparison between programs, with respect to execution result and also execution cost. An observational equivalence indexed by the "greater-than-or-equal" preorder $\geq$ can indeed state that replacing a program fragment with another always requires fewer steps in execution of the whole program. By combining the indexed observational equivalence with a cost model of the UAM, one could hopefully prove improvement [Moran and Sands, 1999], which integrates the idea of reduction of execution cost with observational equivalence.

As a related matter, Sec. 5.1.2 discussed a view of the rewrites-first DGoIM as an optimised variant of the UAM. Another future work is to formalise the idea of optimising the UAM, possibly with the notion of improvement. Optimisation of the UAM could give another avenue for exploring the time-space efficiency trade-off of program execution.

## 6.7 Further directions

**Universality of the UAM** The UAM presented in Chap. 3 is dubbed *universal* in the sense of universal algebras. Like universal algebras are parametrised by a set of operations and their equational theory, the UAM is parametrised by a set of

operations and their behaviour, which is given in terms of focussed graph rewriting. One could ask if the UAM is *universal* also in the sense of universal Turing machines, which can simulate arbitrary Turing machines. It would be interesting to see how the UAM can be instantiated to simulate known abstract machines, such as Landin's SECD machine [Landin, 1964] for the lambda-calculus.

**Local-reasoning assistant** Chap. 4 formulated the proof methodology of observational equivalence that exploits locality. An equivalence proof is boiled down to elementary case-by-case analysis of interference between sub-graphs, formalised as the notions of input-safety and robustness. Although analysis of each case is arguably elementary, the main challenge, which can be observed in Sec. 4.5.5, is to identify all possible interference between particular sub-graphs. An approach taken in Sec. 4.5.5 was to focus on paths and labels, as summarised in Table 4.6. Investigating this approach, and in general, approaches to detecting interference, is important future work, which would be an essential aid to local reasoning.

**Type system** Another direction of further research is to equip SPARTAN with a more expressive type system, compared with the current one which merely ensures that terms are formed correctly. More powerful type systems could be used to ensure safety of program execution, by disproving an error state, or to statically trace and analyse behaviour of certain operations, like with type and effect systems [Nielson and Nielson, 1999]. Although these type systems are not necessary for the UAM and its proof methodology of observational equivalence, which can be seen as a strength of the UAM, it would be interesting to study how the notion of typing in SPARTAN can benefit local reasoning.

# Chapter A

# Technical appendix for Chap. 3

## A.1 Equivalent definitions of hypernets

Informally, hypernets are nested hypergraphs, and one hypernet can contain nested hypergraphs up to different depths. This intuition is reflected by Def. 3.3.5 of hypernets, in particular the big union in $\mathcal{H}_{k+1}(L,M) = \mathcal{H}\Big(L, M \cup \bigcup_{i\leq k} \mathcal{H}_i(L,M)\Big)$. In fact, the definition can be replaced by a simpler, but possibly less intuitive, definition below that does not explicitly deal with the different depths of nesting.

**Definition A.1.1.** Given sets $L$ and $M$, a set $\mathcal{H}'_k(L,M)$ is defined by induction on $k \in \mathbb{N}$:

$$\begin{aligned} \mathcal{H}'_0(L,M) &:= \mathcal{H}(L,M) \\ \mathcal{H}'_{k+1}(L,M) &:= \mathcal{H}\Big(L, M \cup \mathcal{H}'_k(L,M)\Big) \end{aligned}$$

and hence a set $\mathcal{H}'_\omega(L,M) := \bigcup_{i\in\mathbb{N}} \mathcal{H}'_i(L,M)$.

**Lemma A.1.2.** *Given arbitrary sets $L$ and $M$, any two numbers $k, k' \in \mathbb{N}$ satisfy $\mathcal{H}'_k(L,M) \subseteq \mathcal{H}'_{k+k'}(L,M)$.*

*Proof.* If $k' = 0$, the inclusion trivially holds. If not, i.e. $k' > 0$, it can be proved by induction on $k \in \mathbb{N}$. The key reasoning principle we use is that $M \subseteq M'$ implies $\mathcal{H}(L,M) \subseteq \mathcal{H}(L,M')$.

In the base case, when $k = 0$ (and $k' > 0$), we have

$$\begin{aligned}\mathcal{H}'_0(L, M) &= \mathcal{H}(L, M) \\ &\subseteq \mathcal{H}\Big(L, M \cup \mathcal{H}'_{k'-1}(L, M)\Big) = \mathcal{H}'_{k'}(L, M).\end{aligned}$$

In the inductive case, when $k > 0$ (and $k' > 0$), we have

$$\begin{aligned}\mathcal{H}'_k(L, M) &= \mathcal{H}\Big(L, M \cup \mathcal{H}'_{k-1}(L, M)\Big) \\ &\subseteq \mathcal{H}\Big(L, M \cup \mathcal{H}'_{k-1+k'}(L, M)\Big) = \mathcal{H}'_{k+k'}(L, M)\end{aligned}$$

where the inclusion is by induction hypothesis on $k - 1$. □

**Proposition A.1.3.** *Any sets $L$ and $M$ satisfy $\mathcal{H}_k(L, M) = \mathcal{H}'_k(L, M)$ for any $k \in \mathbb{N}$, and hence $\mathcal{H}_\omega(L, M) = \mathcal{H}'_\omega(L, M)$.*

*Proof.* We first prove $\mathcal{H}_k(L, M) \subseteq \mathcal{H}'_k(L, M)$ by induction on $k \in \mathbb{N}$. The base case, when $k = 0$, is trivial. In the inductive case, when $k > 0$, we have

$$\begin{aligned}\mathcal{H}_k(L, M) &= \mathcal{H}\Big(L, M \cup \bigcup_{i \leq k-1} \mathcal{H}_i(L, M)\Big) \\ &\subseteq \mathcal{H}\Big(L, M \cup \bigcup_{i \leq k-1} \mathcal{H}'_i(L, M)\Big) && \text{(by I.H.)} \\ &= \mathcal{H}\Big(L, M \cup \mathcal{H}'_{k-1}(L, M)\Big) && \text{(by Lem. A.1.2)} \\ &= \mathcal{H}'_k(L, M).\end{aligned}$$

The other direction, i.e. $\mathcal{H}'_k(L, M) \subseteq \mathcal{H}_k(L, M)$, can be also proved by induction on $k \in \mathbb{N}$. The base case, when $k = 0$, is again trivial. In the inductive case, we

have

$$\begin{aligned}
\mathcal{H}'_k(L,M) &= \mathcal{H}\Big(L, M \cup \mathcal{H}'_{k-1}(L,M)\Big) \\
&\subseteq \mathcal{H}\Big(L, M \cup \mathcal{H}_{k-1}(L,M)\Big) && \text{(by I.H.)} \\
&\subseteq \mathcal{H}\Big(L, M \cup \bigcup_{i \leq k-1} \mathcal{H}_i(L,M)\Big) \\
&= \mathcal{H}_k(L,M).
\end{aligned}$$

□

Given a hypernet $G$, by Lem. A.1.2 and Prop. A.1.3, there exists a minimum number $k$ such that $G \in \mathcal{H}'_k(L,M)$, which we call the *minimum level* of $G$.

**Lemma A.1.4.** *Any hypernet has a finite number of shallow edges, and a finite number of deep edges.*

*Proof.* Any hypernet has a finite number of shallow edges by definition. We prove that any hypernet $G$ has a finite number of deep edges, by induction on minimum level $k$ of the hypernet.

When $k = 0$, the hypernet has ho deep edges.

When $k > 0$, each hypernet $H$ that labels a shallow edge of $G$ belongs to $\mathcal{H}'_{k-1}(L,M)$, and therefore its minimum level is less than $k$. By induction hypothesis, the labelling hypernet $H$ has a finite number of deep edges, and also a finite number of shallow edges. Deep edges of $G$ are given by edges, at any depth, of any hypernet that labels a shallow edge of $G$. Because there is a finite number of the hypernets that label the shallow edges of $G$, the number of deep edges of $G$ is finite. □

## A.2 Plugging

An interfaced labelled monoidal hypergraph can be given by data of the following form: $((V \uplus I \uplus O, E), (S, T), (f_{\mathsf{V}}, f_{\mathsf{E}}))$ where $I$ is the input list, $O$ is the output list, $V$ is the set of all the other vertices, $E$ is the set of edges, $(S, T)$ defines source and target lists, and $(f_{\mathsf{V}}, f_{\mathsf{E}})$ is labelling functions.

**Definition A.2.1** (Plugging)**.** Let $\mathcal{C}[\vec{\chi^1}, \chi, \vec{\chi^2}] = ((V \uplus I \uplus O, E), (S, T), (f_{\mathsf{V}}, f_{\mathsf{E}}))$ and $\mathcal{C}'[\vec{\chi^3}] = ((V' \uplus I' \uplus O', E'), (S', T'), (f'_{\mathsf{V}}, f'_{\mathsf{E}}))$ be contexts, such that the hole $\chi$ and the latter context $\mathcal{C}'$ have the same type and $\vec{\chi^1} \cap \vec{\chi^2} \cap \vec{\chi^3} = \emptyset$. The *plugging* $\mathcal{C}[\vec{\chi^1}, \mathcal{C}', \vec{\chi^2}]$ is a hypernet given by data $((\hat{V}, \hat{E}), (\hat{S}, \hat{T}), (\hat{f}_{\mathsf{V}}, \hat{f}_{\mathsf{E}}))$ such that:

$$\hat{V} = V \uplus V' \uplus I \uplus O$$

$$\hat{E} = (E \backslash \{e_\chi\}) \uplus E'$$

$$\hat{S}(e) = \begin{cases} S(e) & (\text{if } e \in E \backslash \{e_\chi\}) \\ g^*(S'(e)) & (\text{if } e \in E') \end{cases}$$

$$\hat{T}(e) = \begin{cases} T(e) & (\text{if } e \in E \backslash \{e_\chi\}) \\ g^*(T'(e)) & (\text{if } e \in E') \end{cases}$$

$$g(v) = \begin{cases} v & (\text{if } v \in V') \\ (S(e_\chi))_i & (\text{if } v = (I')_i) \\ (T(e_\chi))_i & (\text{if } v = (O')_i) \end{cases}$$

$$\hat{f}_{\mathsf{V}}(v) = \begin{cases} f_{\mathsf{V}}(v) & (\text{if } v \in V) \\ f'_{\mathsf{V}}(v) & (\text{if } v \in V') \end{cases}$$

$$\hat{f}_{\mathsf{E}}(e) = \begin{cases} f_{\mathsf{E}}(e) & (\text{if } e \in E \backslash \{e_\chi\}) \\ f'_{\mathsf{E}}(e) & (\text{if } e \in E') \end{cases}$$

where $e_\chi \in E$ is the hole edge labelled with $\chi$, and $(-)_i$ denotes the $i$-th element of a list.

In the resulting context $\mathcal{C}[\vec{\chi'}, \mathcal{C}', \vec{\chi''}]$, each edge comes from either $\mathcal{C}$ or $\mathcal{C}'$. If a path in $\mathcal{C}$ does not contain the hole edge $e_\chi$, the path gives a path in $\mathcal{C}[\vec{\chi'}, \mathcal{C}', \vec{\chi''}]$. Conversely, if a path in $\mathcal{C}[\vec{\chi'}, \mathcal{C}', \vec{\chi''}]$ consists of edges from $\mathcal{C}$ only, the path gives a path in $\mathcal{C}$.

Any path in $\mathcal{C}'$ gives a path in $\mathcal{C}[\vec{\chi'}, \mathcal{C}', \vec{\chi''}]$. However, if a path in $\mathcal{C}[\vec{\chi'}, \mathcal{C}', \vec{\chi''}]$ consists of edges from $\mathcal{C}'$ only, the path does not necessarily give a path in $\mathcal{C}'$. The path indeed gives a path in $\mathcal{C}'$, if sources and targets of the hole edge $e_\chi$ are distinct in $\mathcal{C}$ (i.e. the hole edge $e_\chi$ is not a self-loop).

## A.3 Rooted states

**Lemma A.3.1.** *Let $(X, \twoheadrightarrow)$ is an abstract rewriting system that is deterministic.*

1. *For any $x, y, y' \in X$ such that $y$ and $y'$ are normal forms, and for any $k, h \in \mathbb{N}$, if there exist two sequences $x \twoheadrightarrow^k y$ and $x \twoheadrightarrow^h y'$, then these sequences are exactly the same.*

2. *For any $x, y \in X$ such that $y$ is a normal form, and for any $i, j, k \in \mathbb{N}$ such that $i \neq j$ and $i, j \in \{1, \ldots, k\}$, if there exists a sequence $x \twoheadrightarrow^k y$, then its $i$-th rewrite $z \twoheadrightarrow z'$ and $j$-th rewrite $w \twoheadrightarrow w'$ satisfy $z \neq w$.*

*Proof.* The point (1) is proved by induction on $k + h \in \mathbb{N}$. In the base case, when $k + h = 0$ (i.e. $k = h = 0$), the two sequences are both the empty sequence, and $x = y = y'$. The inductive case, when $k + h > 0$, falls into one of the following two situations. The first situation, where $k = 0$ or $h = 0$, boils down to the base case, because $x$ must be a normal form itself, which means $k = h = 0$. In the second situation, where $k > 0$ and $h > 0$, there exist elements $z, z' \in X$ such that $x \twoheadrightarrow z \twoheadrightarrow^{k-1} y$ and $x \twoheadrightarrow z' \twoheadrightarrow^{h-1} y'$. Because $\twoheadrightarrow$ is deterministic, $z = z'$ follows,

and hence by induction hypothesis on $(k-1)+(h-1)$, these two sequences are the same.

The point (2) is proved by contradiction. The sequence $x \rightarrowtriangle^k y$ from $x$ to the normal form $y$ is unique, by the point (1). If its $i$-th rewrite $z \rightarrowtriangle z'$ and $j$-th rewrite $w \rightarrowtriangle w'$ satisfy $z = w$, determinism of the system implies that these two rewrites are the same. This means that the sequence $x \rightarrowtriangle^k y$ has a cyclic sub-sequence, and by repeating the cycle different times, one can yield different sequences of rewrites $x \rightarrowtriangle^* y$ from $x$ to $y$. This contradicts the uniqueness of the original sequence $x \rightarrowtriangle^k y$. □

**Lemma A.3.2.** *If a state $\dot{G}$ is rooted, a search sequence $?;|\dot{G}| \overset{\bullet}{\rightarrow}{}^* \dot{G}$ from the initial state $?;|\dot{G}|$ to the state $\dot{G}$ is unique. Moreover, for any $i$-th search transition and $j$-th search transition in the sequence such that $i \neq j$, these transitions do not result in the same state.*

*Proof.* Let $X$ be the set of states with search or value token. We can define an abstract rewriting system $(X, \rightarrowtriangle)$ of "reverse search" by: $\dot{H} \rightarrowtriangle \dot{H}'$ if $\dot{H}' \overset{\bullet}{\rightarrow} \dot{H}$. Any search sequence corresponds to a sequence of rewrites in this rewriting system.

The rewriting system is deterministic, i.e. if $\dot{H}' \overset{\bullet}{\rightarrow} \dot{H}$ and $\dot{H}'' \overset{\bullet}{\rightarrow} \dot{H}$ then $\dot{H}' = \dot{H}''$, because the inverse $\mapsto^{-1}$ of the interaction rules (Fig. 3.5) is deterministic.

If a search transition changes a token to a search token, the resulting search token always has an incoming operation edge. This means that, in the rewriting system $(X, \rightarrowtriangle)$, initial states are normal forms. Therefore, by Lem. A.3.1(1), if there exist two search sequences from the initial state $?;|\dot{G}|$ to the state $\dot{G}$, these search sequences are exactly the same. The rest is a consequence of Lem. A.3.1(2). □

**Lemma A.3.3.** *For any hypernet $N$, if there exists an operation path from an input to a vertex, the path is unique. Moreover, no edge appears twice in the operation path.*

*Proof.* Given the hypernet $N$ whose set of (shallow) vertices is $X$, we can define an

abstract rewriting system $(X, \twoheadrightarrow)$ of "reverse connection" by: $v \twoheadrightarrow v'$ if there exists an operation edge whose unique source is $v'$ and targets include $v$. Any operation path from an input to a vertex in $N$ corresponds to a sequence of rewrites in this rewriting system.

This rewriting system is deterministic, because each vertex can have at most one incoming edge in a hypergraph (Def. 3.3.2) and each operation edge has exactly one source. Because inputs of the hypernet $N$ have no incoming edges, they are normal forms in this rewriting system. Therefore, by Lem. A.3.1(1), an operation path from any input to any vertex is unique.

The rest is proved by contradiction. We assume that, in an operation path $P$ from an input to a vertex, the same operation edge $e$ appears twice. The edge $e$ has one source, which either is an input of the hypernet $N$ or has an incoming edge. In the former case, the edge $e$ can only appear as the first edge of the operation path $P$, which is a contradiction. In the latter case, the operation edge $e$ has exactly one incoming edge $e'$ in the hypernet $N$. In the operation path $P$, each appearance of the operation edge $e$ must be preceded by this edge $e'$ via the same vertex. This contradicts Lem. A.3.1(2). □

**Lemma A.3.4.** *For any rooted state $\dot{G}$, if its token source (i.e. the source of the token) does not coincide with the unique input, then there exists an operation path from the input to the token source.*

*Proof.* By Lem. A.3.2, the rooted state $\dot{G}$ has a unique search sequence $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \dot{G}$. The proof is by the length $k$ of this sequence.

In the base case, where $k = 0$, the state $\dot{G}$ itself is an initial state, which means the input and token source coincide in $\dot{G}$.

In the inductive case, where $k > 0$, there exists a state $\dot{G}'$ such that $?; |\dot{G}| \xrightarrow{\bullet}{}^{k-1} \dot{G}' \xrightarrow{\bullet} \dot{G}$. The proof here is by case analysis on the interaction rule used in $\dot{G}' \xrightarrow{\bullet} \dot{G}$.

- When the interaction rules (1a), (1b), (2) or (5b) is used (see Fig. 3.5), the

transition $\dot{G}' \overset{\bullet}{\to} \dot{G}$ only changes a token label.

- When the interaction rule (3) is used, the transition $\dot{G}' \overset{\bullet}{\to} \dot{G}$ turns the token and its outgoing operation edge $e_{G'}$ into an operation edge $e_G$ and its outgoing token. By induction hypothesis on $\dot{G}'$, the token source coincides with its input, or there exists an operation path from the input to the token source, in $\dot{G}'$.

  In the former case, in $\dot{G}$, the source of the operation edge $e_G$ coincides with the input. The edge $e_G$ itself gives the desired operation path in $\dot{G}$.

  In the latter case, the operation path $P_{G'}$ from the input to the token source in $\dot{G}'$ does not contain the outgoing operation edge $e_{G'}$ of the token; otherwise, the edge $e_{G'}$ must be preceded by the token edge in the operation path $P_{G'}$, which is a contradiction. Therefore, the operation path $P_{G'}$ in $\dot{G}'$ is inherited in $\dot{G}$, becoming a path $P_G$ from the input to the source of the incoming operation edge $e_G$ of the token. In the state $\dot{G}$, the path $P_G$ followed by the edge $e_G$ yields the desired operation path.

- When the interaction rule (4) is used, the transition $\dot{G}' \overset{\bullet}{\to} \dot{G}$ changes the token from a $(k+1)$-th outgoing edge of an operation edge $e$ to a $(k+2)$-th outgoing edge of the same operation edge $e$, for some $k \in \mathbb{N}$. In $\dot{G}'$, the token source is not an input, and therefore, there exists an operation path $P_{G'}$ from the input to the token source, by induction hypothesis.

  The operation path $P_{G'}$ ends with the operation edge $e$, and no outgoing edge of the edge $e$ is involved in the path $P_{G'}$; otherwise, the edge $e$ must appear more than once in the path $P_{G'}$, which is a contradiction by Lem. A.3.3. Therefore, the path $P_{G'}$ is inherited exactly as it is in $\dot{G}$, and it gives the desired operation path.

- When the interaction rule (5a) is used, by the same reasoning as in the case

of rule (4), $\dot{G}'$ has an operation path $P_{G'}$ from the input to the token source, where the incoming operation edge $e_{G'}$ of the token appears exactly once, at the end. Removing the edge $e_{G'}$ from the path $P_{G'}$ yields another operation path $P$ from the input in $\dot{G}'$, and it also gives an operation path from the input to the token source in $\dot{G}$.

□

**Lemma A.3.5.** *For any state $\dot{G}$ with a $\mathtt{t}$-token such that $\mathtt{t} \neq ?$, if $\dot{G}$ is rooted, then there exists a search sequence $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G} \rangle_{?/\mathtt{t}} \xrightarrow{\bullet}{}^{+} \dot{G}$.*

*Proof.* By Lem. A.3.2, the rooted state $\dot{G}$ has a unique search sequence $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \dot{G}$. The proof is to show that a transition from the state $\langle \dot{G} \rangle_{?/\mathtt{t}}$ appears in this search seqeunce, and it is by the length $k$ of the search sequence.

Because $\dot{G}$ does not have a search token, $k = 0$ is impossible, and therefore the base case is when $k = 1$. The search transition $?; |\dot{G}| \xrightarrow{\bullet} \dot{G}$ must use one of the interaction rules (1a), (1b), (2) and (5b). This means $?; |\dot{G}| = \langle \dot{G} \rangle_{?/\mathtt{t}}$.

In the inductive case, where $k > 0$, there exists a state $\dot{G}'$ such that $?; |\dot{G}| \xrightarrow{\bullet}{}^{k-1} \dot{G}' \xrightarrow{\bullet} \dot{G}$. The proof here is by case analysis on the interaction rule used in $\dot{G}' \xrightarrow{\bullet} \dot{G}$.

- When the interaction rule (1a), (1b), (2) or (5b) is used, $?; |\dot{G}| = \langle \dot{G} \rangle_{?/\mathtt{t}}$.
- Because $\dot{G}$ does not have a search token, the interaction rules (3) and (4) can be never used in $\dot{G}' \xrightarrow{\bullet} \dot{G}$.
- When the interaction rule (5a) is used, $\dot{G}'$ has a value token, which is a $(k+1)$-th outgoing edge of an operation edge $e$, for some $k \in \mathbb{N}$. The operation edge $e$ becomes the outgoing edge of the token in $\dot{G}$. By induction hypothesis on $\dot{G}'$, we have

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G}' \rangle_{?/\checkmark} \xrightarrow{\bullet}{}^{+} \dot{G}' \xrightarrow{\bullet} \dot{G}. \tag{A}$$

  If $k = 0$, in $\dot{G}'$, the token is the only outgoing edge of the operation edge $e$. Because $\langle \dot{G}' \rangle_{?/\checkmark}$ is not an initial state, it must be a result of the interaction

rule (3), which means the search sequence (A) is factored through as:

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G} \rangle_{?/\mathtt{t}} \xrightarrow{\bullet} \langle \dot{G}' \rangle_{?/\checkmark} \xrightarrow{\bullet}{}^{+} \dot{G}' \xrightarrow{\bullet} \dot{G}.$$

If $k > 0$, for each $m \in \{0, \ldots, k\}$, let $\dot{N}_m$ be a state with a search token, such that $|\dot{N}_m| = |\dot{G}'|$ and the token is an $(m+1)$-th outgoing edge of the operation edge $e$. This means $\dot{N}_k = \langle \dot{G}' \rangle_{?/\checkmark}$. The proof concludes by combining the following internal lemma with (A), taking $k$ as $m$.

**Lemma A.3.6.** *For any $m \in \{0, \ldots, k\}$, if there exists $h < k$ such that $?; |\dot{G}| \xrightarrow{\bullet}{}^{h} \dot{N}_m$, then it is factored through as $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G} \rangle_{?/\mathtt{t}} \xrightarrow{\bullet}{}^{+} \dot{N}_m$.*

*Proof.* By induction on $m$. In the base case, when $m = 0$, the token of $\dot{N}_m$ is the first outgoing edge of the operation edge $e$. This state is not initial, and therefore must be a result of the interaction rule (3), which means

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G} \rangle_{?/\mathtt{t}} \xrightarrow{\bullet} \dot{N}_m.$$

In the inductive case, when $m > 0$, the state $\dot{N}_m$ is not an initial state and must be a result of the interaction rule (4), which means

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{N_{m-1}} \rangle_{\checkmark/?} \xrightarrow{\bullet} \dot{N}_m.$$

The first half of this search sequence, namely $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{N_{m-1}} \rangle_{\checkmark/?}$, consists of $h - 1 < k$ transitions. Therefore, by (outer) induction hypothesis on $h - 1$, we have

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \dot{N_{m-1}} \xrightarrow{\bullet}{}^{+} \langle \dot{N_{m-1}} \rangle_{\checkmark/?} \xrightarrow{\bullet} \dot{N}_m.$$

The first part, namely $?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \dot{N_{m-1}}$, consists of less than $k$ transitions.

Therefore, by (inner) induction hypothesis on $m-1$, we have

$$?; |\dot{G}| \xrightarrow{\bullet}{}^{*} \langle \dot{G} \rangle_{?/\mathtt{t}} \xrightarrow{\bullet}{}^{+} \dot{N_{m-1}} \xrightarrow{\bullet}{}^{+} \langle \dot{N_{m-1}} \rangle_{\checkmark/?} \xrightarrow{\bullet} \dot{N_m}.$$

□

□

**Lemma A.3.7.**

1. *For any state $\dot{N}$, if it has a path to the token source that is not an operation path, then it is not rooted.*

2. *For any focus-free hypernet $H$ and any focussed context $\dot{\mathcal{C}}[\chi]$ with one hole edge, such that $\dot{\mathcal{C}}[H]$ is a state, if the hypernet $H$ is one-way and the context $\dot{\mathcal{C}}$ has a path to the token source that is not an operation path, then the state $\dot{\mathcal{C}}[H]$ is not rooted.*

3. *For any $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\vec{\chi}]; \vec{G}; \vec{H})$ of an output-closed pre-tepmlate $\lhd$, if the context $\dot{\mathcal{C}}[\vec{\chi}]$ has a path to the token source that is not an operation path, then at least one of the states $\dot{\mathcal{C}}[\vec{G}]$ and $\dot{\mathcal{C}}[\vec{H}]$ is not rooted.*

*Proof of the point (1).* Let $P$ be the path in $\dot{N}$ to the token source that is not an operation path. The proof is by contradiction; we assume that $\dot{N}$ is a rooted state.

Because of $P$, the token source is not an input. Therefore by Lem. A.3.4, the state $\dot{N}$ has an operation path from its unique input to the token source. This operation path contradicts the path $P$, which is not an operation path, because each operation edge has only one source and each vertex has at most one incoming edge. □

*Proof of the point (2).* Let $P$ be the path in $\dot{\mathcal{C}}$ to the token source that is not an operation path.

If the path $P$ contains no hole edge, it gives a path in the state $\dot{\mathcal{C}}[H]$ to the token source that is not an operation path. By the point (1), the state is not rooted.

Otherwise, i.e. if the path $P$ contains a hole edge, we give a proof by contradiction; we assume that the state $\dot{\mathcal{C}}[H]$ is rooted. We can take a suffix of the path $P$, so that it gives a path from a target of a hole edge to the token source in $\dot{\mathcal{C}}$, and moreover, gives a path $P'$ from a source of an edge from $H$ to the token source in $\dot{\mathcal{C}}[H]$. This implies the token source is not an input, and therefore by Lem. A.3.4, the state $\dot{\mathcal{C}}[H]$ has an operation path from its unique input to the token source. This operation path must have $P'$ has a suffix, meaning $P'$ is also an operation path, because each operation edge has only one source and each vertex has at most one incoming edge. Moreover, $H$ must have an operation path from an input to an output, such that the input and the output have type $\star$ and the path ends with the first edge of the path $P'$. This contradicts $H$ being one-way. □

*Proof of the point (3).* Let $P$ be the path in $\dot{\mathcal{C}}$ to the token source that is not an operation path.

If the path $P$ contains no hole edge, it gives a path in the states $\dot{\mathcal{C}}[\vec{G}]$ and $\dot{\mathcal{C}}[\vec{H}]$ to the token source that is not an operation path. By the point (1), the states are not rooted.

Otherwise, i.e. if the path $P$ contains a hole edge, we can take a suffix of $P$ that gives a path $P'$ from a source of a hole edge $e$ to the token source in $\dot{\mathcal{C}}$, so that the path $P'$ does not contain any hole edge. We can assume that the hole edge $e$ is labelled with $\chi_1$, without loss of generality. The path $P'$ gives paths $P'_G$ and $P'_H$ to the token source, in contexts $\dot{\mathcal{C}}[\chi_1, \vec{G}\backslash\{G_1\}]$ and $\dot{\mathcal{C}}[\chi_1, \vec{H}\backslash\{H_1\}]$, respectively. The paths $P'_G$ and $P'_H$ are not an operation path, because they start with the hole edge $e$ labelled with $\chi_1$.

Because $\lhd$ is output-closed, $G_1$ or $H_1$ is one-way. By the point (2), at least one of the states $\dot{\mathcal{C}}[\vec{G}]$ and $\dot{\mathcal{C}}[\vec{H}]$ is not rooted. □

**Lemma A.3.8.** *If a rewrite transition $\dot{G} \to \dot{G}'$ is stationary, it preserves the rooted property, i.e. $\dot{G}$ being rooted implies $\dot{G}'$ is also rooted.*

*Proof.* The stationary rewrite transition $\dot{G} \to \dot{G}'$ is in the form of $\mathcal{C}[\lightning ;_i H] \to \mathcal{C}[?;_i H']$, where $\mathcal{C}$ is a focus-free simple context, $H$ is a focus-free one-way hypernet, $H'$ is a focus-free hypernet and $i \in \mathbb{N}$. We assume $\mathcal{C}[\lightning ;_i H]$ is rooted, and prove that $\mathcal{C}[?;_i H']$ is rooted, i.e. $?;\mathcal{C}[H'] \overset{\bullet}{\to}{}^* \mathcal{C}[?;_i H']$. By Lem. A.3.5, there exists a number $k \in \mathbb{N}$ such that:

$$?;\mathcal{C}[H] \overset{\bullet}{\to}{}^k \mathcal{C}[?;_i H] \overset{\bullet}{\to}{}^+ \mathcal{C}[\lightning ;_i H].$$

The rest of the proof is by case analysis on the number $k$.

- When $k = 0$, i.e. $?;\mathcal{C}[H] = \mathcal{C}[?;_i H]$, the unique input and the $i$-th source of the hole coincide in the simple context $\mathcal{C}$. Therefore, $?;\mathcal{C}[H'] = \mathcal{C}[?;_i H']$, which means $\mathcal{C}[?;_i H']$ is rooted.

- When $k > 0$, there exists a state $\dot{N}$ such that $?;\mathcal{C}[H] \overset{\bullet}{\to}{}^{k-1} \dot{N} \overset{\bullet}{\to} \mathcal{C}[?;_i H]$. By the following internal lemma (Lem. A.3.9), there exists a focussed simple context $\dot{\mathcal{C}_N}$, whose token is not entering nor exiting, and we have two search sequences:

$$?;\mathcal{C}[H] \overset{\bullet}{\to}{}^{k-1} \dot{\mathcal{C}_N}[H] \overset{\bullet}{\to} \mathcal{C}[?;_i H],$$
$$?;\mathcal{C}[H'] \overset{\bullet}{\to}{}^{k-1} \dot{\mathcal{C}_N}[H'].$$

  The last search transition $\dot{\mathcal{C}_N}[H] \overset{\bullet}{\to} \mathcal{C}[?;_i H]$, which yields a search token, must use the interaction rule (3) or (4). Because the token is not entering nor exiting in the simple context $\dot{\mathcal{C}_N}$, either of the two interaction rules acts on the token and an edge of the context. This means that the same interaction is possible in the state $\dot{\mathcal{C}_N}[H']$, yielding:

$$?;\mathcal{C}[H'] \overset{\bullet}{\to}{}^{k-1} \dot{\mathcal{C}_N}[H'] \overset{\bullet}{\to} \mathcal{C}[?;_i H'],$$

which means $\mathcal{C}[?;_i H']$ is rooted.

**Lemma A.3.9.** *For any $m \in \{0, \ldots, k-1\}$ and any state $\dot{N}$ such that $?;\mathcal{C}[H] \xrightarrow{\bullet}{}^{m} \dot{N} \xrightarrow{\bullet}{}^{k-m} \mathcal{C}[?;_i H]$, the following holds.*

*(A) If there exists a focussed simple context $\dot{\mathcal{C}_N}$ such that $\dot{N} = \dot{\mathcal{C}_N}[H]$, the token of the context $\dot{\mathcal{C}_N}$ is not entering.*

*(B) If there exists a focussed simple context $\dot{\mathcal{C}_N}$ such that $\dot{N} = \dot{\mathcal{C}_N}[H]$, the token of the context $\dot{\mathcal{C}_N}$ is not exiting.*

*(C) There exists a focussed simple context $\dot{\mathcal{C}_N}$ such that $\dot{N} = \dot{\mathcal{C}_N}[H]$, and $?;\mathcal{C}[H'] \xrightarrow{\bullet}{}^{m} \dot{\mathcal{C}_N}[H']$ holds.*

*Proof.* Firstly, because search transitions do not change an underlying hyper-net, if there exists a focussed simple context $\dot{\mathcal{C}_N}$ such that $\dot{N} = \dot{\mathcal{C}_N}[H]$, $|\dot{\mathcal{C}_N}| = \mathcal{C}$ necessarily holds.

The point (A) is proved by contradiction; we assume that the context $\dot{\mathcal{C}_N}$ has an entering token. This means that there exist a number $p \in \mathbb{N}$ and a token label $\mathtt{t} \in \{?, \checkmark, \lightning\}$ such that $\dot{\mathcal{C}_N} = \mathcal{C}[\mathtt{t};_p H]$. By Lem. A.3.5, there exists a number $h$ such that $h \leq m$ and:

$$?;\mathcal{C}[H] \xrightarrow{\bullet}{}^{h} \mathcal{C}[?;_p H] \xrightarrow{\bullet}{}^{k-h} \mathcal{C}[?;_i H]. \qquad (\$)$$

We derive a contradiction by case analysis on the numbers $p$ and $h$.

- If $p = i$ and $h = 0$, the state $\mathcal{C}[?;_i H]$ must be initial, but it is a result of a search transition because $k - h > 0$. This is a contradiction.
- If $p = i$ and $h > 0$, two different transitions in the search sequence (\$) result in the same state, because of $h > 0$ and $k-h > 0$, which contradicts Lem. A.3.2.

– If $p \neq i$, by Def. 4.3.2, there exists a state $\dot{N}'$ with a rewrite token such that $\mathcal{C}[?;_p H] \xrightarrow{\bullet} \dot{N}'$. This contradicts the search sequence (\$), because $k - h > 0$ and search transitions are deterministic.

The point (B) follows from the contraposition of Lem. A.3.7(2), because $H$ is one-way and $\dot{N}$ is rooted. The rooted property of $\dot{N}$ follows from the fact that search transitions do not change underlying hypernets.

The point (C) is proved by induction on $m \in \{0, \ldots, k-1\}$. In the base case, when $m = 0$, we have $?;\mathcal{C}[H] = \dot{N}$, and therefore the context $?;\mathcal{C}$ can be taken as $\dot{\mathcal{C}}_N$. This means $?;\mathcal{C}[H'] = \dot{\mathcal{C}}_N[H']$.

In the inductive case, when $m > 0$, there exists a state $\dot{N}'$ such that

$$?;\mathcal{C}[H] \xrightarrow{\bullet}{}^{m-1} \dot{N}' \xrightarrow{\bullet} \dot{N} \xrightarrow{\bullet}{}^{k-m} \mathcal{C}[?;_i H].$$

By the induction hypothesis, there exists a focussed simple context $\dot{\mathcal{C}}_{N'}$ such that $\dot{N}' = \dot{\mathcal{C}}_{N'}[H]$ and

$$?;\mathcal{C}[H] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}_{N'}[H] \xrightarrow{\bullet} \dot{N} \xrightarrow{\bullet}{}^{k-m} \mathcal{C}[?;_i H],$$
$$?;\mathcal{C}[H'] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}_{N'}[H'].$$

Our goal here is to find a focussed simple context $\dot{\mathcal{C}}_N$, such that $\dot{N} = \dot{\mathcal{C}}_N[H]$ and $\dot{\mathcal{C}}_{N'}[H'] \xrightarrow{\bullet} \dot{\mathcal{C}}_N[H']$.

In the search transition $\dot{\mathcal{C}}_{N'}[H] \xrightarrow{\bullet} \dot{N}$, the only change happens to the token and its incoming or outgoing edge $e$ in the state $\dot{\mathcal{C}}_{N'}[H]$. By the points (A) and (B), the token is not entering nor exiting in the context $\dot{\mathcal{C}}_{N'}$, which means the edge $e$ must be from the context, not from $H$.

Now that no edge from $H$ is changed in $\dot{\mathcal{C}}_{N'}[H] \xrightarrow{\bullet} \dot{N}$, there exists a focussed simple context $\dot{\mathcal{C}}_N$ such that $\dot{N} = \dot{\mathcal{C}}_N[H]$, and moreover, $\dot{\mathcal{C}}_{N'}[H'] \xrightarrow{\bullet} \dot{\mathcal{C}}_N[H']$. □

□

## A.4 Accessible paths and stable hypernets

A stable hypernet always has at least one edge, and any non-output vertex is labelled with $\star$. It has a tree-like shape.

**Lemma A.4.1** (Shape of Stable Hypernets)**.**

1. *In any stable hypernet, if a vertex $v'$ is reachable from another vertex $v$ such that $v \neq v'$, there exists a unique path from the vertex $v$ to the vertex $v'$.*

2. *Any stable hypernet has no cyclic path, i.e. a path from a vertex to itself.*

3. *Let $\mathcal{C} : \star \Rightarrow \otimes_{i=1}^{m} \ell_i$ be a simple context such that: its hole has one source and at least one outgoing edge; and its unique input is the hole's source. There are no two stable hypernets $G$ and $G'$ that satisfy $G = \mathcal{C}[G']$.*

*Proof.* To prove the point (1), assume there are two different paths from the vertex $v$ to the vertex $v'$. These paths, i.e. non-empty sequences of edges, have to involve an edge with more than one source, or two different edges that share the same target. However, neither of these is possible in a stable hypernet, because both a passive operation edge and an instance edge have only one source and vertices can have at most one incoming edge. The point (1) follows from this by contradiction.

If a stable hypernet has a cyclic path from a vertex $v$ to itself, there must be infinitely many paths from the input to the vertex $v$, depending on how many times the cycle is included. This contradicts the point (1).

The point (3) is also proved by contradiction. Assume that there exist two stable hypernets $G$ and $G'$ that satisfy $G = \mathcal{C}[G']$ for the simple context $\mathcal{C}$. In the stable hypernet $G$, a vertex is always labelled with $\star$ if it is not an output. However, in the simple context $\mathcal{C}$, there exists at least one target of the hole that is not an output

of the context but not labelled with $\star$ either. This contradicts $\mathcal{C}[G']$ being a stable hypernet. □

A stable hypernet can be found as a part of representation of a value.

**Lemma A.4.2.** *Let $\vec{x}$ be a sequence of $k$ variables and $\vec{a}$ be a sequence of $h$ atoms. For any derivable type judgement $\vec{x} \mid \vec{a} \vdash v : \star$ where $v$ is a value, its representation can be decomposed as $(\vec{x} \mid \vec{a} \vdash v : \star)^\dagger = \mathcal{C}[G]$ using a stable hypernet $G : \star \Rightarrow \otimes_{i=1}^{m} \ell_i$ and a simple context $\mathcal{C} : \star \Rightarrow \star^{\otimes k} \otimes \diamond^{\otimes h}$ whose unique input coincides with a (unique) source of its hole.*

*Proof.* By induction on the definition of value.

When the value $v$ is an atom, in the representation $(\vec{x} \mid \vec{a} \vdash v : \star)^\dagger$, only an instance edge can comprise a stable hypernet.

When the value is $v \equiv \phi_\checkmark(v_1, \ldots, v_m; \vec{s})$, by induction hypothesis, a stable hypernet $G_i$ can be extracted from (a bottom part of) representation of each eager argument $v_i$. The stable hypernet $G$ that decomposes the representation $(\vec{x} \mid \vec{a} \vdash v : \star)^\dagger$ can be given by all these stable hypernets $G_1, \ldots, G_m$ together with the passive operation edge $\phi_\checkmark$ that is introduced in the representation.

When the value is $v \equiv \texttt{bind}\ x \to t\ \texttt{in}\ v'$, or $v \equiv \texttt{new}\ a \multimap t\ \texttt{in}\ v'$, by induction hypothesis, representation of the value $v'$ includes a stable hypernet $G'$. The stable hypernet itself decomposes the representation $(\vec{x} \mid \vec{a} \vdash v : \star)^\dagger$ in the required way. □

**Lemma A.4.3.** *For any state $\dot{N}$, and its vertex $v$, such that the vertex $v$ is not a target of an instance edge or a passive operation edge, if an accessible path from the vertex $v$ is stable or active, then the path has no multiple occurences of a single edge.*

*Proof.* Any stable or active path consists of edges that has only one source. As a consequence, except for the first edge, no edge appears twice in the stable path. If the stable path is from the vertex $v$, its first edge also does not appear twice, because $v$ is not a target of an instance edge or a passive operation edge. □

**Lemma A.4.4.** *For any state $\dot{N}$, and its vertex $v$, such that the vertex $v$ is not a target of an instance edge or a passive operation edge, the following are equivalent.*

*(A) There exist a focussed simple context $\dot{\mathcal{C}}[\chi]$ and a stable hypernet $G$, such that $\dot{N} = \dot{\mathcal{C}}[G]$, where the vertex $v$ of $\dot{N}$ corresponds to a unique source of the hole edge in $\dot{\mathcal{C}}$.*

*(B) Any accessible path from the vertex $v$ in $\dot{N}$ is a stable path.*

*Proof of (A) $\Rightarrow$ (B).* Because no output of a stable hypernet has type $\star$, any path from the vertex $v$ in $\dot{\mathcal{C}}[G]$ gives a path from the unique input in $G$. In the stable hypernet $G$, any path from the unique input is a stable path. □

*Proof of (B) $\Rightarrow$ (A).* In the state $\dot{N}$, the token target has to be a source of an edge, which forms an accessible path itself. By Lem. A.4.3, in the state $\dot{N}$, we can take maximal stable paths from the vertex $v$, in the sense that appending any edge to these paths, if possible, does not give a stable path.

If any of these maximal stable paths is to some vertex, the vertex does not have type $\star$; this can be confirmed as follows. If the vertex has type $\star$, it is not an output, so it is a source of an instance, token, operation or contraction edge. The case of an instance or passive operation edge contradicts the maximality. The other case yields a non-stable accessible path that contradicts the assumption (B).

Collecting all edges contained by the maximal stable paths, therefore, gives the desired hypernet $G$. These edges are necessarily all shallow, because of the vertex $v$ of $\dot{N}$. The focussed context $\dot{\mathcal{C}}[\chi]$, whose hole is shallow, can be made of all the other edges (at any depth) of the state $\dot{N}$. □

**Lemma A.4.5.** *Let $\dot{N}$ be a state, where the token is an incoming edge of an operation edge $e$, whose label $\phi$ takes at least one eager arguments. Let $k$ denote the number of eager arguments of $\phi$.*

*For each $i \in \{1, \ldots, k\}$, let $sw_i(\dot{N})$ be a state such that: both states $sw_i(\dot{N})$ and $\dot{N}$ have the same token label and the same underlying hypernet, and the token in*

*$sw_i(\dot{N})$ is the $i$-th outgoing edge of the operation edge $e$.*

*For each $i \in \{1, \ldots, k\}$, the following are equivalent.*

*(A) In $\dot{N}$, any accessible path from an $i$-th target of the operation edge $e$ is a stable (resp. active) path.*

*(B) In $sw_i(\dot{N})$, any accessible path from the token target is a stable (resp. active) path.*

*Proof.* The only difference between $\dot{N}$ and $sw_i(\dot{N})$ is the swap of the token with the operation edge $e$, and these two edges form an accessible path in the states $\dot{N}$ and $sw_i(\dot{N})$, individually or together (in an appropriate order). Therefore, there is one-to-one correspondence between accessible paths from an $i$-th target of the edge $e$ in $\dot{N}$, and accessible paths from the token target in $sw_i(\dot{N})$.

When (A) is the case, in $\dot{N}$, any accessible paths from an $i$-th target of the edge $e$ does not contain the token nor the edge $e$; otherwise there would be an accessible path that contains the token and hence not stable nor active, which is a contradiction. This means that, in $sw_i(\dot{N})$, any accessible path from the token target also does not contain the token nor the edge $e$, and the path must be a stable (resp. active) path.

When (B) is the case, the proof takes the same reasoning in the reverse way. □

**Lemma A.4.6.** *Let $\dot{N}$ be a rooted state with a search token, such that the token is not an incoming edge of a contraction edge.*

1. *$\dot{N} \xrightarrow{\bullet}{}^{+} \langle \dot{N} \rangle_{\checkmark/?}$, if and only if any accessible path from the token target in $\dot{N}$ is a stable path.*

2. *$\dot{N} \xrightarrow{\bullet}{}^{+} \langle \dot{N} \rangle_{\lightning/?}$, if and only if any accessible path from the token target in $\dot{N}$ is an active path.*

*Proof of the forward direction.* Let $\mathtt{t}$ be either '$\checkmark$' or '$\lightning$'. The assumption is $\dot{N} \xrightarrow{\bullet}{}^{*} \langle \dot{N} \rangle_{\mathtt{t}/?}$. We prove the following, by induction on the length $n$ of this search sequence:

- any accessible path from the token target in $\dot{N}$ is a stable path, when $\mathtt{t} = \checkmark$, and
- any accessible path from the token target in $\dot{N}$ is an active path, when $\mathtt{t} = \lightning$.

In the base case, where $n = 1$, because the token is not an incoming edge of a contraction edge, the token target is a source of an instance edge, or an operation edge labelled with $\phi \in \mathbb{O}_{\mathtt{t}}$ that takes no eager argument. In either situation, the outgoing edge of the token gives the only possible accessible path from the token target. The path is stable when $\mathtt{t} = \checkmark$, and active when $\mathtt{t} = \lightning$.

In the inductive case, where $n > 1$, the token target is a source of an operation edge $e_\phi$ labelled with an operation $\phi \in \mathbb{O}_{\mathtt{t}}$ that takes at least one eager argument.

Let $k$ denote the number of eager arguments of $\phi_{\mathtt{t}}$, and $i$ be an arbitrary number in $\{1, \ldots, k\}$. Let $sw_i(\dot{N})$ be the state as defined in Lem. A.4.5. Because $\dot{N}$ is rooted, by Lem. A.3.5, the given search sequence gives the following search sequence (proof by induction on $k - i$):

$$?; |\dot{N}| \xrightarrow{\bullet}{}^{*} \dot{N} \xrightarrow{\bullet}{}^{+} sw_i(\dot{N}) \xrightarrow{\bullet}{}^{+} \langle sw_i(\dot{N}) \rangle_{\checkmark/?} \xrightarrow{\bullet}{}^{+} \langle \dot{N} \rangle_{\mathtt{t}/?}.$$

By induction hypothesis on the intermediate sequence $sw_i(\dot{N}) \xrightarrow{\bullet}{}^{+} \langle sw_i(\dot{N}) \rangle_{\checkmark/?}$, any accessible path from the token target in $sw_i(\dot{N})$ is a stable path. By Lem. A.4.5, any accessible path from an $i$-th target of the operation edge $e_\phi$ in $\dot{N}$ is a stable path.

In $\dot{N}$, any accessible path from the token target is given by the operation edge $e_\phi$ followed by an accessible path, which is proved to be stable above, from a target of $e_\phi$. Any accessible path from the token target is therefore stable when $\mathtt{t} = \checkmark$, and active when $\mathtt{t} = \lightning$. □

*Proof of the backward direction.* Let $\mathtt{t}$ be either '$\checkmark$' or '$\lightning$'. The assumption is the following:

- any accessible path from the token target in $\dot{N}$ is a stable path, when $\mathtt{t} = \checkmark$, and
- any accessible path from the token target in $\dot{N}$ is an active path, when $\mathtt{t} = \lightning$.

Our goal is to show $\dot{N} \xrightarrow{\bullet}{}^{*} \langle \dot{N} \rangle_{\mathtt{t}/?}$.

In the state $\dot{N}$, the token target has to be a source of an edge, which forms an accessible path itself. By Lem. A.4.3, we can define $r(\dot{N})$ by the maximum length of stable paths from the token target. This number $r(\dot{N})$ is well-defined and positive. We prove $\dot{N} \xrightarrow{\bullet}{}^{*} \langle \dot{N} \rangle_{\mathtt{t}/?}$ by induction on $r(\dot{N})$.

In the base case, where $r(\dot{N}) = 1$, the outgoing edge of the token is the only possible accessible path from the token target. The outgoing edge is not a contraction edge by the assumption, and hence it is an instance edge, or an operation edge labelled with $\phi \in \mathbb{O}_{\mathtt{t}}$ that takes no eager argument. We have $\dot{N} \xrightarrow{\bullet} \langle \dot{N} \rangle_{\mathtt{t}/?}$.

In the inductive case, where $r(\dot{N}) > 1$, the outgoing edge of the token is an operation edge $e_\phi$ labelled with $\phi \in \mathbb{O}_{\mathtt{t}}$ that takes at least one eager argument. Any accessible path from the token target in $\dot{N}$ is given by the edge $e_\phi$ followed by a stable path from a target of $e_\phi$.

Let $k$ denote the number of eager arguments of $\phi_{\mathtt{t}}$, and $i$ be an arbitrary number in $\{1, \ldots, k\}$. Let $sw_i(\dot{N})$ be the state as defined in Lem. A.4.5.

By the assumption, any accessible path from an $i$-th target of the operation edge $e_\phi$ in $\dot{N}$ is a stable path. Therefore by Lem. A.4.5, in $sw_i(\dot{N})$, any accessible path from the token target is a stable path. Moreover, these paths in $\dot{N}$ and $sw_i(\dot{N})$ correspond to each other. By Lem. A.4.3, we can define $r(sw_i(\dot{N}))$ by the maximum length of stable paths from the token target. This number $r(sw_i(\dot{N}))$ is well-defined, and satisfies $r(sw_i(\dot{N})) < r(\dot{N})$. By induction hypothesis on this number, we have:

$$sw_i(\dot{N}) \xrightarrow{\bullet}{}^{*} \langle sw_i(\dot{N}) \rangle_{\checkmark/?}.$$

Combining this search sequence with the following possible search transitions con-

cludes the proof:

$$\dot{N} \xrightarrow{\bullet} sw_1(\dot{N}),$$

$$\langle sw_i(\dot{N})\rangle_{\checkmark/?} \xrightarrow{\bullet} sw_{i+1}(\dot{N}),$$

(when $k \neq 1$ and $i < k$)

$$\langle sw_k(\dot{N})\rangle_{\checkmark/?} \xrightarrow{\bullet} \langle \dot{N}\rangle_{\mathtt{t}/?}.$$

□

## A.5 Parametrised (contextual) refinement and equivalence

**Lemma A.5.1.** *For any focus-free contexts $\mathcal{C}_1[\vec{\chi'}, \chi, \vec{\chi''}]$ and $\mathcal{C}_2$ such that $\mathcal{C}_1[\vec{\chi'}, \mathcal{C}_2, \vec{\chi''}]$ is defined, if both $\mathcal{C}_1$ and $\mathcal{C}_2$ are binding-free, then $\mathcal{C}_1[\vec{\chi'}, \mathcal{C}_2, \vec{\chi''}]$ is also binding-free.*

*Proof.* Let $\mathcal{C}$ denote $\mathcal{C}_1[\vec{\chi'}, \mathcal{C}_2, \vec{\chi''}]$, and $e_\chi$ denote the hole edge of $\mathcal{C}_1$ labelled with $\chi$.

The proof is by contradiction. We assume that there exists a path $P$ in $\mathcal{C}$, from a source of a contraction, atom, box or hole edge $e$, to a source of a hole edge $e'$. We derive a contradiction by case analysis on the path $P$.

- When $e'$ comes from $\mathcal{C}_1$, and the path $P$ consists of edges from $\mathcal{C}_1$ only, the path $P$ gives a path in $\mathcal{C}_1$ that contradicts $\mathcal{C}_1$ being binding-free.

- When $e'$ comes from $\mathcal{C}_1$, and the path $P$ contains an edge from $\mathcal{C}_2$, by finding the last edge from $\mathcal{C}_2$ in $P$, we can take a suffix of $P$ that gives a path from a target of the hole edge $e_\chi$ to a source of a hole edge, in $\mathcal{C}_1$. Adding the hole edge $e_\chi$ at the beginning yields a path in $\mathcal{C}_1$ that contradicts $\mathcal{C}_1$ being binding-free.

- When both $e$ and $e'$ come from $\mathcal{C}_2$, and the path $P$ gives a path in $\mathcal{C}_2$, this contradicts $\mathcal{C}_2$ being binding-free.

- When both $e$ and $e'$ come from $\mathcal{C}_2$, and the path $P$ does not give a single path in $\mathcal{C}_2$, there exists a path from a source of the hole edge $e_\chi$ to a source of the hole edge $e_\chi$, in $\mathcal{C}_1$. This path contradicts $\mathcal{C}_1$ being binding-free.

- When $e$ comes from $\mathcal{C}_1$ and $e'$ comes from $\mathcal{C}_2$, by finding the first edge from $\mathcal{C}_2$ in $P$, we can take a prefix of $P$ that gives a path from a source of a contraction, atom, box or hole edge to a source of the hole edge $e_\chi$, in $\mathcal{C}_1$. This path contradicts $\mathcal{C}_1$ being binding-free.

□

**Lemma A.5.2.** *For any set $\mathbb{C}$ of contexts that is closed under plugging, and any preorder $Q$ on natural numbers, the following holds.*

- *$\dot{\preceq}_Q$ and $\preceq_Q^{\mathbb{C}}$ are reflexive.*

- *$\dot{\preceq}_Q$ and $\preceq_Q^{\mathbb{C}}$ are transitive.*

- *$\dot{\simeq}_Q$ and $\simeq_Q^{\mathbb{C}}$ are equivalences.*

*Proof.* Because $\dot{\simeq}_Q$ and $\simeq_Q^{\mathbb{C}}$ are defined as a symmetric subset of $\dot{\preceq}_Q$ and $\preceq_Q^{\mathbb{C}}$, respectively, $\dot{\simeq}_Q$ and $\simeq_Q^{\mathbb{C}}$ are equivalences if $\dot{\preceq}_Q$ and $\preceq_Q^{\mathbb{C}}$ are preorders.

Reflexivity and transitivity of $\dot{\preceq}_Q$ is a direct consequence of those of the preorder $Q$.

For any focus-free hypernet $H$, and any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$ such that $?;\mathcal{C}[H]$ is a state, $?;\mathcal{C}[H] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H]$ because of reflexivity of $\dot{\preceq}_Q$.

For any focus-free hypernets $H_1$, $H_2$ and $H_3$, and any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$, such that $H_1 \preceq_Q^{\mathbb{C}} H_2$, $H_2 \preceq_Q^{\mathbb{C}} H_3$, and both $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_3]$ are states, our goal is to show $?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_3]$. Because $H_1 \preceq_Q^{\mathbb{C}} H_2$ and $H_2 \preceq_Q^{\mathbb{C}} H_3$, all three hypernets $H_1$, $H_2$ and $H_3$ have the same type, and hence $?;\mathcal{C}[H_2]$ is also a state. Therefore, we have $?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_2]$ and $?;\mathcal{C}[H_2] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_3]$, and the transitivity of $\dot{\preceq}_Q$ implies $?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_3]$. □

**Lemma A.5.3.** *For any set $\mathbb{C}$ of contexts that is closed under plugging, and any preorder $Q$ on natural numbers, the following holds.*

1. *For any hypernets $H_1$ and $H_2$, $H_1 \simeq^{\mathbb{C}}_{Q \cap Q^{-1}} H_2$ implies $H_1 \simeq^{\mathbb{C}}_{Q} H_2$.*
2. *If all compute transitions are deterministic, for any hypernets $H_1$ and $H_2$, $H_1 \simeq^{\mathbb{C}}_{Q} H_2$ implies $H_1 \simeq^{\mathbb{C}}_{Q \cap Q^{-1}} H_2$.*

*Proof.* Because $(Q \cap Q^{-1}) \subseteq Q$, the point (1) follows from the monotonicity of contextual equivalence.

For the point (2), $H_1 \simeq^{\mathbb{C}}_{Q} H_2$ means that any focus-free context $\mathcal{C}[\chi] \in \mathbb{C}$, such that $?;\mathcal{C}[H_1]$ and $?;\mathcal{C}[H_2]$ are states, yields $?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_2]$ and $?;\mathcal{C}[H_2] \mathrel{\dot{\preceq}_Q} ?;\mathcal{C}[H_1]$. If the state $?;\mathcal{C}[H_1]$ terminates at a final state after $k_1$ transitions, there exists $k_2$ such that $k_1 \ Q \ k_2$ and the state $?;\mathcal{C}[H_2]$ terminates at a final state after $k_2$ transitions. Moreover, there exists $k_3$ such that $k_2 \ Q \ k_3$ and the state $?;\mathcal{C}[H_1]$ terminates at a final state after $k_3$ transitions.

Because search transitions and copy transitions are deterministic, if all compute transitions are deterministic, states and transitions comprise a deterministic abstract rewriting system, in which final states are normal forms. By Lem. A.3.1, $k_1 = k_3$ must hold. This means $k_1 \ Q \cap Q^{-1} \ k_2$, and $?;\mathcal{C}[H_1] \mathrel{\dot{\preceq}_{Q \cap Q^{-1}}} ?;\mathcal{C}[H_2]$. Similarly, we can infer $?;\mathcal{C}[H_2] \mathrel{\dot{\preceq}_{Q \cap Q^{-1}}} ?;\mathcal{C}[H_1]$, and hence $H_1 \simeq^{\mathbb{C}}_{Q \cap Q^{-1}} H_2$. □

## A.6 Proof for Sec. 4.3.3

**Lemma A.6.1.** *Let $\mathbb{C}$ be a set of contexts, and $Q'$ be a binary relation on $\mathbb{N}$ such that, for any $k_0, k_1, k_2 \in \mathbb{N}$, $(k_0 + k_1) \ Q' \ (k_0 + k_2)$ implies $k_1 \ Q' \ k_2$. Let $\lhd$ be a pre-template that is a trigger and implies contextual refinement $\preceq^{\mathbb{C}}_{Q'}$. For any single $\mathbb{C}$-specimen $(\dot{\mathcal{C}}[\chi]; H^1; H^2)$ of $\lhd$, the following holds.*

1. *For any $k \in \mathbb{N}$, $?;|\dot{\mathcal{C}}|[H^1] \overset{\bullet}{\to}^k \dot{\mathcal{C}}[H^1]$ if and only if $?;|\dot{\mathcal{C}}|[H^2] \overset{\bullet}{\to}^k \dot{\mathcal{C}}[H^2]$.*

2. *If compute transitions are all deterministic, and one of states* $\dot{\mathcal{C}}[H^1]$ *and* $\dot{\mathcal{C}}[H^2]$ *is rooted, then the other state is also rooted, and moreover,* $\dot{\mathcal{C}}[H^1] \mathrel{\dot{\preceq}_{Q'}} \dot{\mathcal{C}}[H^2]$.

*Proof of the point (1).* Let $(p, q)$ be an arbitrary element of a set $\{(1,2),(2,1)\}$. We prove that, for any $k \in \mathbb{N}$, $?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{k} \dot{\mathcal{C}}[H^p]$ implies $?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{k} \dot{\mathcal{C}}[H^q]$. The proof is by case analysis on the number $k$.

- When $k = 0$, $\dot{\mathcal{C}}[H^p]$ is initial, and by Lem. 4.4.5(1), $\dot{\mathcal{C}}[H^q]$ is also initial. Note that $\lhd$ is a trigger and hence output-closed.

- When $k > 0$, by the following internal lemma, $?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{k} \dot{\mathcal{C}}[H^q]$ follows from $?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{k} \dot{\mathcal{C}}[H^p]$.

  **Lemma A.6.2.** *For any* $m \in \{0, \ldots, k\}$*, there exists a focussed context* $\dot{\mathcal{C}}'[\chi]$ *such that* $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$ *and the following holds:*

$$?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{m} \dot{\mathcal{C}}'[H^p] \xrightarrow{\bullet}{}^{k-m} \dot{\mathcal{C}}[H^p],$$
$$?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{m} \dot{\mathcal{C}}'[H^q].$$

  *Proof.* By induction on $m$. In the base case, when $m = 0$, we can take $?; |\dot{\mathcal{C}}|$ as $\dot{\mathcal{C}}'$.

  In the inductive case, when $m > 0$, by induction hypothesis, there exists a focussed context $\dot{\mathcal{C}}'[\chi]$ such that $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$ and the following holds:

$$?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^p] \xrightarrow{\bullet}{}^{k-m+1} \dot{\mathcal{C}}[H^p],$$
$$?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^q].$$

  Because $|\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}| \in \mathbb{C}$, $(\dot{\mathcal{C}}'; H^1; H^2)$ is a single $\mathbb{C}$-specimen of $\lhd$, which yields rooted states. Because $k - m + 1 > 0$, $\dot{\mathcal{C}}'$ cannot have a rewrite token. The rest of the proof is by case analysis on the token of $\dot{\mathcal{C}}'$.

– When $\dot{\mathcal{C}}'$ has an entering search token, because $\lhd$ is a trigger, $\dot{\mathcal{C}}'[H^r] \to \langle\dot{\mathcal{C}}'[H^r]\rangle_{\lightning/?}$ for each $r \in \{p, q\}$. Because $\langle\dot{\mathcal{C}}'[H^r]\rangle_{\lightning/?} = \langle\dot{\mathcal{C}}'\rangle_{\lightning/?}[H^r]$, and search transitions are deterministic, we have the following:

$$?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^p] \xrightarrow{\bullet} \langle\dot{\mathcal{C}}'\rangle_{\lightning/?}[H^p] \xrightarrow{\bullet}{}^{k-m} \dot{\mathcal{C}}[H^p],$$
$$?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^q] \xrightarrow{\bullet} \langle\dot{\mathcal{C}}'\rangle_{\lightning/?}[H^q].$$

We also have $|\langle\dot{\mathcal{C}}'\rangle_{\lightning/?}| = |\dot{\mathcal{C}}'| = |\dot{\mathcal{C}}|$.

– When $\dot{\mathcal{C}}'$ has a value token, or a non-entering search token, because $\lhd$ is output-closed, by Lem. 4.4.5(3), there exists a focussed context $\dot{\mathcal{C}}''$ such that $|\dot{\mathcal{C}}''| = |\dot{\mathcal{C}}'|$ and $\dot{\mathcal{C}}'[H^r] \to \dot{\mathcal{C}}''[H^r]$ for each $r \in \{p, q\}$. The transition $\dot{\mathcal{C}}'[H^r] \to \dot{\mathcal{C}}''[H^r]$, for each $r \in \{p, q\}$, is a search transition, and by the determinism of search transitions, we have the following:

$$?; |\dot{\mathcal{C}}|[H^p] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^p] \xrightarrow{\bullet} \dot{\mathcal{C}}''[H^p] \xrightarrow{\bullet}{}^{k-m} \dot{\mathcal{C}}[H^p],$$
$$?; |\dot{\mathcal{C}}|[H^q] \xrightarrow{\bullet}{}^{m-1} \dot{\mathcal{C}}'[H^q] \xrightarrow{\bullet} \dot{\mathcal{C}}''[H^q].$$

□

□

*Proof of the point (2).* If one of states $\dot{\mathcal{C}}[H^1]$ and $\dot{\mathcal{C}}[H^2]$ is rooted, by the point (1), the other state is also rooted, and moreover, there exists $k \in \mathbb{N}$ such that $?; |\dot{\mathcal{C}}|[H^r] \xrightarrow{\bullet}{}^{k} \dot{\mathcal{C}}[H^r]$ for each $r \in \{1, 2\}$.

Our goal is to prove that, for any $k_1 \in \mathbb{N}$ and any final state $\dot{N}_1$ such that $\dot{\mathcal{C}}[H^1] \to^{k_1} \dot{N}_1$, there exist $k_2 \in \mathbb{N}$ and a final state $\dot{N}_2$ such that $k_1 \; Q' \; k_2$ and

$\dot{\mathcal{C}}[H^2] \rightarrow^{k_2} \dot{N}_2$. Assuming $\dot{\mathcal{C}}[H^1] \rightarrow^{k_1} \dot{N}_1$, we have the following:

$$?; |\dot{\mathcal{C}}|[H^1] \stackrel{\bullet}{\rightarrow}^{k} \dot{\mathcal{C}}[H^1] \rightarrow^{k_1} \dot{N}_1,$$
$$?; |\dot{\mathcal{C}}|[H^2] \stackrel{\bullet}{\rightarrow}^{k} \dot{\mathcal{C}}[H^2].$$

Because $\lhd$ implies contextual refinement $\preceq^{\mathbb{C}}_{Q'}$, and $|\dot{\mathcal{C}}| \in \mathbb{C}$, we have state refinement $?; |\dot{\mathcal{C}}|[H^1] \dot{\preceq}_{Q'} ?; |\dot{\mathcal{C}}|[H^2]$. Therefore, there exist $l_2 \in \mathbb{N}$ and a final state $\dot{N}_2$ such that $(k + k_1)\ Q'\ l_2$ and $?; |\dot{\mathcal{C}}|[H^2] \rightarrow^{l_2} \dot{N}_2$.

The assumption that compute transitions are all deterministic implies that all transitions, including intrinsic ones, are deterministic. Following from this are $l_2 \geq k$ and the following:

$$?; |\dot{\mathcal{C}}|[H^1] \stackrel{\bullet}{\rightarrow}^{k} \dot{\mathcal{C}}[H^1] \rightarrow^{k_1} \dot{N}_1,$$
$$?; |\dot{\mathcal{C}}|[H^2] \stackrel{\bullet}{\rightarrow}^{k} \dot{\mathcal{C}}[H^2] \rightarrow^{l_2 - k} \dot{N}_2.$$

By the assumption on $Q'$, $(k + k_1)\ Q'\ l_2$ implies $k_1\ Q'\ (l_2 - k)$. □

# Bibliography

S. Abramsky. *The lazy lambda-calculus*, page 65–117. Addison Wesley, 1990. ISBN 0-201-17236-4. (pages 6 and 8).

S. Abramsky and G. McCusker. Call-by-value games. In *CSL 1997*, volume 1414 of *Lect. Notes Comp. Sci.*, pages 1–17. Springer, 1998. doi: 10.1007/BFb0028004. (pages 8 and 9).

S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS 1998*, pages 334–344. IEEE Computer Society, 1998. doi: 10.1109/LICS.1998.705669. (pages 8 and 9).

S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. doi: 10.1006/inco.2000.2930. (pages 6 and 8).

B. Accattoli. The complexity of abstract machines. In *WPTE 2016*, volume 235 of *EPTCS*, pages 1–15, 2017. doi: 10.4204/EPTCS.235.1. (pages 4, 11, 20, 47, and 51).

B. Accattoli and B. Barras. Environments and the complexity of abstract machines. In *PPDP 2017*, pages 4–16. ACM, 2017. doi: 10.1145/3131851.3131855. (page 171).

B. Accattoli and U. Dal Lago. (Leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Comp. Sci.*, 12(1), 2016. doi: 10.2168/LMCS-12(1:4)2016. (page 4).

B. Accattoli and S. Guerrini. Jumping boxes. In *CSL 2009*, volume 5771 of *Lect. Notes Comp. Sci.*, pages 55–70. Springer, 2009. doi: 10.1007/978-3-642-04027-6_7. (pages 28 and 173).

B. Accattoli and D. Kesner. The structural lambda-calculus. In *CSL 2010*, volume 6247 of *Lect. Notes Comp. Sci.*, pages 381–395. Springer, 2010. doi: 10.1007/978-3-642-15205-4_30. (pages 21 and 52).

B. Accattoli and C. Sacerdoti Coen. On the value of variables. In *WoLLIC 2014*, volume 8652 of *Lect. Notes Comp. Sci.*, pages 36–50. Springer, 2014. doi: 10.1007/978-3-662-44145-9_3. (pages 47 and 52).

B. Accattoli, P. Barenbaum, and D. Mazza. Distilling abstract machines. In *ICFP 2014*, pages 363–376. ACM, 2014. doi: 10.1145/2628136.2628154. (pages 4, 20, 47, and 52).

A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP 2006*, volume 3924 of *Lect. Notes Comp. Sci.*, pages 69–83. Springer, 2006. doi: 10.1007/11693024_6. (page 8).

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL 2009*, pages 340–353. ACM, 2009. doi: 10.1145/1480881.1480925. (pages 8 and 90).

A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi: 10.1145/504709.504712. (page 8).

F. Bonchi, F. Gadducci, A. Kissinger, P. Soboci'nski, and F. Zanasi. Rewriting modulo symmetric monoidal structure. In *LICS 2016*, pages 710–719. ACM, 2016. doi: 10.1145/2933575.2935316. (page 174).

F. Bonchi, D. Petrisan, D. Pous, and J. Rot. A general account of coinduction up-to. *Acta Inf.*, 54(2):127–190, 2017. doi: 10.1007/s00236-016-0271-4. (page 117).

S. W. T. Cheung, V. Darvariu, D. R. Ghica, K. Muroya, and R. N. S. Rowe. A functional perspective on machine learning via programmable induction and abduction. In *FLOPS 2018*, volume 10818 of *Lect. Notes Comp. Sci.*, pages 84–98. Springer, 2018. doi: 10.1007/978-3-319-90686-7_6. (pages 18, 163, and 165).

B. Coecke and R. Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, 2011. doi: 10.1088/1367-2630/13/4/043016. (page 93).

P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007. doi: 10.1007/s10990-007-9015-z. (page 172).

U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *LICS 2011*, pages 133–142. IEEE Computer Society, 2011. doi: 10.1109/LICS.2011.22. (page 12).

U. Dal Lago and B. Petit. Linear dependent types in a call-by-value scenario. In *PPDP 2012*, pages 115–126. ACM, 2012. doi: 10.1145/2370776.2370792. (page 12).

U. Dal Lago and U. Schöpp. Computation by interaction for space-bounded functional programming. *Inf. Comput.*, 248:150–194, 2016. doi: 10.1016/j.ic.2015.04.006. (page 12).

U. Dal Lago, C. Faggian, B. Valiron, and A. Yoshimizu. Parallelism and synchronization in an infinitary context. In *LICS 2015*, pages 559–572. IEEE Computer Society, 2015. doi: 10.1109/LICS.2015.58. (page 60).

U. Dal Lago, C. Faggian, B. Valiron, and A. Yoshimizu. The geometry of parallelism: classical, probabilistic, and quantum effects. In *POPL 2017*, pages 833–845. ACM, 2017. doi: 10.1145/3009837. (pages 60 and 175).

V. Danos and L. Regnier. Reversible, irreversible and optimal lambda-machines. *Elect. Notes in Theor. Comp. Sci.*, 3:40–60, 1996. doi: 10.1016/S1571-0661(05)80402-5. (pages xi, 12, 58, 59, 61, and 62).

O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990. doi: 10.1145/91556.91622. (page 72).

O. Danvy and I. Zerny. A synthetic operational account of call-by-need evaluation. In *PPDP 2013*, pages 97–108. ACM, 2013. doi: 10.1145/2505879.2505898. (pages 20, 50, and 172).

O. Danvy, K. Millikin, J. Munk, and I. Zerny. On inter-deriving small-step and big-step semantics: a case study for storeless call-by-need evaluation. *Theor. Comp. Sci.*, 435:21–42, 2012. doi: 10.1016/j.tcs.2012.02.023. (pages 21 and 106).

F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002. doi: 10.1006/jcss.2001.1790. (page 174).

D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012. doi: 10.1017/S095679681200024X. (pages 8 and 9).

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comp. Sci.*, 103(2):235–271, 1992. doi: 10.1016/0304-3975(92)90014-7. (page 81).

M. Fernández and I. Mackie. Call-by-value lambda-graph rewriting without rewriting. In *ICGT 2002*, volume 2505 of *LNCS*, pages 75–89. Springer, 2002. doi: 10.1007/3-540-45832-8_8. (pages 60, 61, and 172).

M. Fernández and N. Siafakas. New developments in environment machines. *Elect. Notes in Theor. Comp. Sci.*, 237:57–73, 2009. doi: 10.1016/j.entcs.2009.03.035. (page 172).

M. Fernández, I. Mackie, and F. Sinot. Closed reduction: explicit substitutions without alpha-conversion. *Math. Struct. in Comp. Sci.*, 15(2):343–381, 2005. doi: 10.1017/S0960129504004633. (page 172).

D. R. Ghica. Slot games: a quantitative model of computation. In *POPL 2005*, pages 85–97. ACM, 2005. doi: 10.1145/1040305.1040313. (page 10).

D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *POPL 2007*, pages 363–375. ACM, 2007. doi: 10.1145/1190216.1190269. (page 12).

D. R. Ghica and A. I. Smith. Geometry of synthesis III: resource management through type inference. In *POPL 2011*, pages 345–356. ACM, 2011. doi: 10.1145/1926385.1926425. (page 12).

D. R. Ghica, A. I. Smith, and S. Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In *ICFP 2011*, pages 221–233. ACM, 2011. doi: 10.1145/2034773.2034805. (page 12).

J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. (pages 25, 27, 159, 160, and 173).

J.-Y. Girard. Geometry of Interaction I: interpretation of system F. In *Logic Colloquium 1988*, volume 127 of *Studies in Logic & Found. Math.*, pages 221–260. Elsevier, 1989. doi: 10.1016/S0049-237X(08)70271-4. (page 11).

G. Gonthier, M. Abadi, and J. Lévy. The geometry of optimal lambda reduction. In *POPL 1992*, pages 15–26. ACM Press, 1992. doi: 10.1145/143165.143172. (page 177).

J. Hackett and G. Hutton. Worker/wrapper/makes it/faster. In *ICFP 2014*, pages 95–107. ACM, 2014. doi: 10.1145/2628136.2628142. (page 10).

J. Hackett and G. Hutton. Parametric polymorphism and operational improvement. *PACMPL*, 2(ICFP):68:1–68:24, 2018. doi: 10.1145/3236763. (page 10).

N. Hoshino, K. Muroya, and I. Hasuo. Memoryful Geometry of Interaction: from coalgebraic components to algebraic effects. In *CSL-LICS 2014*, pages 52:1–52:10. ACM, 2014. doi: 10.1145/2603088.2603124. (page 60).

J. M. E. Hyland and C. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. doi: 10.1006/inco.2000.2917. (pages 6 and 8).

P. Johann, A. Simpson, and J. Voigtländer. A generic operational metatheory for algebraic effects. In *LICS 2010*, pages 209–218. IEEE Computer Society, 2010. doi: 10.1109/LICS.2010.29. (page 176).

A. Kissinger. *Pictures of processes: automated graph rewriting for monoidal categories and applications to quantum computing*. PhD thesis, University of Oxford, 2012. arXiv preprint arXiv:1203.0202. (pages 26, 35, and 174).

V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL 2006*, pages 141–152. ACM, 2006. doi: 10.1145/1111037.1111050. (page 8).

V. Koutavas, P. Levy, and E. Sumii. From applicative to environmental bisimulation. *Elect. Notes in Theor. Comp. Sci.*, 276:215–235, 2011. doi: 10.1016/j.entcs.2011.09.023. (page 9).

J. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007. doi: 10.1007/s10990-007-9018-9. (page 61).

Y. Lafont. Interaction nets. In *POPL 1990*, pages 95–108. ACM Press, 1990. doi: 10.1145/96709.96718. (page 173).

Y. Lafont. *From proof nets to interaction nets*, page 225–248. London Mathematical Society Lecture Note Series. Cambridge University Press, 1995. doi: 10.1017/CBO9780511629150.012. (page 173).

J. Laird. Full abstraction for functional languages with control. In *LICS 1997*, pages 58–67. IEEE Computer Society, 1997. doi: 10.1109/LICS.1997.614931. (pages 8 and 9).

J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL 1990*, pages 16–30. ACM Press, 1990. doi: 10.1145/96709.96711. (page 177).

P. Landin. The mechanical evaluation of expressions. *The Comp. Journ.*, 6(4): 308–320, 1964. doi: 10.1093/comjnl/6.4.308. (pages 2, 172, and 178).

J. Lévy. *Optimal reductions in the lambda-calculus*, pages 159–191. Academic Press, 1980. ISBN 0123490502. (page 177).

I. Mackie. The Geometry of Interaction machine. In *POPL 1995*, pages 198–208. ACM, 1995. doi: 10.1145/199448.199483. (pages xi, 12, 58, and 59).

I. Mackie. Linear logic *With* boxes. In *LICS 1998*, pages 309–320. IEEE Computer Society, 1998. doi: 10.1109/LICS.1998.705667. (page 173).

J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comp. Sci.*, 228(1-2):175–210, 1999. doi: 10.1016/S0304-3975(98)00358-2. (pages 39 and 159).

P. Melliès. Functorial boxes in string diagrams. In *CSL 2006*, volume 4207 of *Lect. Notes Comp. Sci.*, pages 1–30. Springer, 2006. doi: 10.1007/11874683_1. (pages 28 and 174).

R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0. (page 117).

E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-66, University of Edinburgh, 1988. (page 175).

A. Moran and D. Sands. Improvement in a lazy context: an operational theory for call-by-need. In *POPL 1999*, pages 43–56. ACM, 1999. doi: 10.1145/292540.292547. (pages 6 and 177).

J. H. Morris Jr. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1969. (page 5).

K. Muroya and D. R. Ghica. The dynamic Geometry of Interaction machine: a call-by-need graph rewriter. In *CSL 2017*, volume 82 of *LIPIcs*, pages 32:1–32:15, 2017. doi: 10.4230/LIPIcs.CSL.2017.32. (pages 17 and 25).

K. Muroya and D. R. Ghica. Efficient implementation of evaluation strategies via token-guided graph rewriting. In *WPTE 2017*, volume 265 of *EPTCS*, pages 52–66, 2018a. doi: 10.4204/EPTCS.265.5. (page 17).

K. Muroya and D. R. Ghica. The dynamic Geometry of Interaction machine: a token-guided graph rewriter. *arXiv preprint arXiv:1803.00427*, 2018b. (page 17).

K. Muroya, N. Hoshino, and I. Hasuo. Memoryful Geometry of Interaction II: recursion and adequacy. In *POPL 2016*, pages 748–760. ACM, 2016. doi: 10.1145/2837614.2837672. (page 60).

K. Muroya, S. W. T. Cheung, and D. R. Ghica. The geometry of computation-graph abstraction. In *LICS 2018*, pages 749–758. ACM, 2018. doi: 10.1145/3209108.3209127. (pages 18, 163, and 165).

F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lect. Notes Comp. Sci.*, pages 114–136. Springer, 1999. doi: 10.1007/3-540-48092-7_6. (page 178).

P. O'Hearn and R. Tennent, editors. *Algol-like languages*. Progress in theoretical computer science. Birkhauser, 1997. doi: 10.1007/978-1-4612-4118-8. (page 90).

C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994. doi: 10.1007/BF01019945. (page 73).

A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In *MFCS 1993*, volume 711 of *Lect. Notes Comp. Sci.*, pages 122–141. Springer, 1993. doi: 10.1007/3-540-57182-5_8. (page 90).

G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, 1973. (pages 6 and 8).

G. D. Plotkin. LCF considered as a programming language. *Theor. Comp. Sci.*, 5 (3):223–255, 1977. doi: 10.1016/0304-3975(77)90044-5. (page 7).

G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, 1981. (page 3).

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004. doi: 10.1016/j.jlap.2004.03.009. (page 3).

G. D. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS 2001*, volume 2030 of *Lect. Notes Comp. Sci.*, pages 1–24. Springer, 2001. doi: 10.1007/3-540-45315-6_1. (page 174).

G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. doi: 10.1023/A:1023064908962. (pages 71 and 174).

G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Comp. Sci.*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013. (page 71).

G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, volume 1: foundations*. World Scientific, 1997. ISBN 9810228848. (page 174).

D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS 2007*, pages 293–302. IEEE Computer Society, 2007. doi: 10.1109/LICS.2007.17. (page 8).

M. Schmidt-Schauß and D. Sabel. Improvements in a call-by-need functional core language: common subexpression elimination and resource preserving translations. *Sci. Comput. Program.*, 147:3–26, 2017. doi: 10.1016/j.scico.2017.01.001. (page 10).

U. Schöpp. Computation-by-interaction with effects. In *APLAS 2011*, volume 7078 of *Lect. Notes Comp. Sci.*, pages 305–321. Springer, 2011. doi: 10.1007/978-3-642-25318-8_23. (page 60).

U. Schöpp. Organising low-level programs using higher types. In *PPDP 2014*, pages 199–210. ACM, 2014a. doi: 10.1145/2643135.2643151. (page 12).

U. Schöpp. Call-by-value in a basic logic for interaction. In *APLAS 2014*, volume 8858 of *Lect. Notes Comp. Sci.*, pages 428–448. Springer, 2014b. doi: 10.1007/978-3-319-12736-1_23. (page 60).

P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997. doi: 10.1017/S0956796897002712. (page 172).

F. Sinot. Call-by-name and call-by-value as token-passing interaction nets. In *TLCA 2005*, volume 3461 of *Lect. Notes Comp. Sci.*, pages 386–400. Springer, 2005. doi: 10.1007/11417170_28. (pages 19, 21, 173, and 177).

F. Sinot. Call-by-need in token-passing nets. *Math. Struct. in Comp. Sci.*, 16(4):639–666, 2006. doi: 10.1017/S0960129506005408. (pages 19, 21, and 173).

R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3):85–97, 1985. doi: 10.1016/S0019-9958(85)80001-2. (pages 6 and 8).

E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. *J. ACM*, 54(5):26, 2007. doi: 10.1145/1284320.1284325. (page 8).

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi: 10.1006/inco.1994.1093. (page 81).

T. Yachi and E. Sumii. A sound and complete bisimulation for contextual equivalence in lambda-calculus with call/cc. In *APLAS 2016*, volume 10017 of *Lect. Notes Comp. Sci.*, pages 171–186, 2016. doi: 10.1007/978-3-319-47958-3_10. (page 9).

# Index