PROOF THEORETIC APPROACH TO SPECIFICATION LANGUAGES

JAWAHAR LAL CHIRIMAR

A DISSERTATION

IN

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1995

Dale Miller

Supervisor of Dissertation

Mark Steedman

Graduate Group Chairman

ACKNOWLWDGEMENTS

I would, first and foremost, like to thank my parents and my wife for their endless support and love during my life as a Ph.D. student.

My committee members, Dale Miller, Carl Gunter, Robert Harper, Jon Riecke, Val Tannen and Peter Buneman provided me with important feedback on this research. I would especially like to thank Dale Miller who encouraged me to think independently and in a scientifically disciplined manner. I learned not only about technical matters from Carl, my supervisor for the first three years, but also about the global picture and the way research is done. From Andre Scedrov, who was on my committee in all but name, I learned the value of rigor – without rigor there is no mathematics and mathematics is not all rigor. I must thank Anil Nerode specially for introducing me to proof theory, functional languages and type theories. I have benefited from various technical discussions with Jean Gallier, Doug Howe, Jim Lipton and Chet Murthy.

There are many people whom I met in my stay at University of Pennsylvania. The numerous discussions with Vijay Gehlot, Ramesh Subramanium, Anuj Dawar, Chuck Liang, Sandeep Biswas, Jon Riecke, Teow Hin Ngair, and other fellow students helped shaped my concepts and opinions. Jon Riecke, Sandeep Biswas and Ernesto Pimentel helped with proof reading this dissertation. Nandini, my wife, helped me clean up the grammar and insert all the appropriate commas and articles, hopefully! The staff of the graduate students office and business office put up very patiently with my eccentric habits and got all my papers through in time.

ABSTRACT

PROOF THEORETIC APPROACH TO SPECIFICATION LANGUAGES

Jawahar Lal Chirimar

Advisor: Dale Miller

In this thesis I study FORUM as a specification language. FORUM is a higher-order logic based on the logical connectives of Linear Logic. As an initial example, I demonstrate that FORUM is well suited for specifying concurrent computations by specifying the higher-order π calculus. Next, I focus on the problem of specifying programming languages with higher-order functions, and imperative features such as assignable variables, exceptions and first-class continuations. I provide a modular and declarative specification of an untyped programming language, UML, which contains the above mentioned features. Further, I use the proof theory of FORUM to study program equivalence for the functional core of UML augmented with assignable variables. Using my compositional specifications in FORUM, I prove equivalence of programs that have been challenging for other specification languages. Finally I study the operation semantics of DLX, a prototypical RISC machine. I specify the sequential and pipelined operational semantics of DLX with important optimizations such as call-forwarding and early branch resolution, and prove them to be equivalent. Furthermore, I study the problem of code equivalence via the FORUM specification, and, in particular, analyze the problem of code rescheduling for DLX.

Contents

1 Introduction **1**

2 FORUM **6**

2.1 FORUM — Logic programming with multiple heads 7

2.2 Specifying $HO\pi$ in FORUM 13

3 Specifying UML **20**

3.1 Λ_v — Functional Core of UML 21

3.2 Λ_{vs} — State in UML 30

3.3 Λ_{ve} — Exceptions in UML 39

3.4 Λ_{vc} — Continuations in UML 51

3.5 UML — Putting it together 58

4 Program Equivalence for Λ_{vs} in FORUM **60**

4.1 Defining Observational Equivalence 61

4.2 Reduction in Λ_{vs} preserves Observational Equivalence 67

4.3 Observational Equivalence proofs in FORUM 72

5 Specifying DLX - a RISC architecture **82**

5.1 The DLX architecture 83

5.2 Sequential specification for DLX architecture 85

5.3 Pipelining DLX - facing the hazards 94

5.4 Call-forwarding and early branch resolution 116

5.5 Program equivalence for DLX — Correctness of code scheduling 127

6 Conclusion and Future Work **132**

6.1 Conclusion 132

6.2 Future Work 134

A Proofs from chapter 3 **138**

Bibliography **153**

List of Figures

2.1 Proof Rules for Forum 10

2.2 Specifying inequality of natural numbers in FORUM 11

2.3 Reduction semantics for $HO\pi$-calculus 15

3.1 Syntax for Λ_v 21

3.2 Natural Semantics specification for Λ_v 23

3.3 Σ_v, Signature for Λ_v evaluator 24

3.4 $\phi_v : Values_v \rightarrow \mathsf{vl}$, $\mathcal{H}_v : \Lambda_v \rightarrow \mathsf{tm}$ 25

3.5 $\psi_v : \mathsf{vl} \rightarrow Values_v$, $\mathcal{L}_v : \mathsf{tm} \rightarrow \Lambda_v$ 25

3.6 Clauses in $\mathcal{E}_v$ - the evaluator for Λ_v 27

3.7 One step in the evaluation of $(M\ N)$ 28

3.8 Syntax for Λ_{vs} 31

3.9 Natural Semantics specification for the new constructs in Λ_{vs} 32

3.10 Constants for translating Λ_{vs} terms 33

3.11 Specification in FORUM for new constructs in Λ_{vs} 34

3.12 Syntax for Λ_{ve} 40

3.13 Natural Semantics specification for the new constructs in Λ_{ve}. 42

3.14 Constants for translating Λ_{ve} terms 45

3.15 Translating the new construct of Λ_{ve} to FORUM 45

3.16 Constants for exception management in FORUM 47

3.17 Specification in FORUM for new constructs in Λ_{ve} 48

3.18 Syntax for Λ_{vc} 53

3.19 Constants for translating Λ_{vc} terms 54

3.20 Specification in FORUM for new constructs in Λ_{vc} 54

3.21 $\mathcal{C}_{vc}$, translation of $EvCont_{vc}$ to FORUM terms of type $\mathsf{vl} \rightarrow o$ 56

3.22 Specification for callcc in the presence of exceptions 57

4.1 Contexts in Λ_{vs} 63

4.2 $\mathcal{C}_{vs}$, translation of $EvCont_{vs}$ to FORUM terms of type $\mathsf{vl} \rightarrow o$ 65

5.1 Semantics of example instructions in DLX. 84

5.2 List of DLX instructions selected for specification. 85

5.3 Block diagram for the connectivity of functional blocks in the DLX. 86

5.4 Signature for specification of DLX 89

5.5 Grammar for DLX programs 90

5.6 Sequential specification of DLX 91

5.7 Example program in DLX 92

5.8 DLX pipeline structure 95

5.9 Signature for specification of DLX 100

5.10 DLX pipeline state transition functions for clock 101

5.11 ρ_1 - table for hazard detection in the DLX pipeline 102

5.12 Specification for the DLX pipeline – clock, bg, IF and ID. 104

5.13 Specification for the DLX pipeline – EX. 105

5.14 Specification for the DLX pipeline – MEM. 106

5.15 Specification for the DLX pipeline – WB. 107

5.16 Changes in the DLX pipeline to reduce branch penalty. 117

5.17 ρ_2 – new table for hazard detection in DLX pipeline 119

5.18 DLX pipeline state transition functions in the presence of call-forwarding and early branch resolution 120

5.19 φ – Call forwarding functions 121

5.20 Specification for the DLX pipeline – IF and ID. 123

5.21 Specification for the DLX pipeline – clock, bg, EX. 124

5.22 Specification for the DLX pipeline – MEM and WB. 125

A.1 Constants in Σ_{ml} used in translating UML to FORUM 139

A.2 Translating UML to FORUM 140

A.3 Translating answers in UML to FORUM 140

A.4 Translating FORUM terms to UML 141

A.5 Matching final configurations of computations in FORUM 145

Chapter 1

Introduction

Ever since we have had programming languages, we have had to *specify the computational behaviors of the languages*. In the most naive sense, we specify the actions of $\mathsf{if}\ \mathsf{M}\ \mathsf{then}\ \mathsf{N}\ \mathsf{else}\ \mathsf{P}$ — a typical phrase in a programming language — by saying that first execute M, and if the result is true then evaluate N , otherwise evaluate P. Even in this simple example, there are many ambiguities inherent in the above specification. For example, it is not clear whether P is to be computed only when M evaluates to false, or even if M evaluates to some other value, say $\mathsf{3}$. On the one hand, we would like a specification to be precise in as much as we would like different implementations of the same specification to have identical computational behavior. On the other hand, if a specification is as precise as an actual implementation of a programming language, then the entire purpose of a specification as a tool to understand the language independently of its implementation is defeated.

Rigorous specifications which define the grammar and meaning of programming languages are indispensable in the present context for a variety of reasons. These include verifying safety of programs, developing optimizing compilers, and maintaining programs as language design evolves. Unfortunately, very few widely used programming languages have such a rigorous specification, the only two I know of being [HMT89, BR90]. In this thesis, I use

FORUM [Mil94] as a specification language. Specifications in FORUM convert computations into proofs — formal objects amenable to logical analysis within the meta-theory of FORUM. I call this style of specification *proof-theoretic* for the key emphasis placed upon proofs and their analysis.

Precise specification provides the sound basis on which one builds implementations of the language and programs. To quote [HMT89] :

> ...for a robust program written in an insecure language is like a house built upon sand.

The main point is that a programming language is different from its implementation — specifications are a way of making precise the behavior that an implementation must exhibit in order to implement a given programming language. Specifications play a crucial role in the present software environment where different commercial vendors are implementing the same language. It will be highly undesirable if programs written in a programming language have different behaviors in two implementations of the same language!

I expect specification languages to play a variety of roles. The most important among these are :

- a specification language should be rich enough to specify imperative, functional and concurrency features modularly
- provide modular specification of different features of the programming language, resulting in better and easier understanding of the programming language itself, and
- should be able to use the meta-theory of the specification language to study the program transformations and verify the correctness of implementations of the specifications.

In this thesis I use FORUM to specify the operational semantics of programming languages. I specify the operational semantics of UML, a prototypical functional language, and the sequential and pipelined operational semantics of the DLX machine [HP90]. I take UML – an untyped higher-order functional language with exceptions, state and callcc, *i.e.* first-class continuations – as a prototypical functional language. UML is the same as the untyped core Standard ML (SML) [HMT89], excluding data-types and pattern matching, augmented with callcc. UML is a significant language in as much as it contains both the functional and imperative features of SML, and thus provides a simpler setting to study the problems that arise due to these features in SML. UML supports first- class continuations because of the rich programming paradigm they provide. The DLX machine is representative of the eminently successful and popular RISC architectures of the last decade including Intel i860, MIPS R2000/R3000, Motorolla 88000, SPARC, PowerPC. The specification of DLX resolves data and structural hazards, and implements optimizations characteristic of modern pipelined machines. The specification of pipelined DLX underlines the fact that FORUM provides the appropriate framework to specify a variety of computational processes – from as high-level and abstract as UML to as low-level and concrete as DLX.

Many attempts have been made at specifying the operational semantics of fragments of UML [Lan64, FF86, WF91, HMT89, MP92, HM92, Han90]. In this thesis I am able to address many issues regarding the specification of UML which the above semantics could not address satisfactorily. Firstly, I am working in a rich meta-theory where concurrency, higher-order functions and imperative features can be specified. This is a mixture of features which has traditionally been very difficult to specify. Secondly, in my translation computations become proofs — formal objects — which I analyze using proof transformations. Using this analysis I am able to prove many program equivalences in [MS88, OT93, SF92, MT92]. Finally, declarative and modular descriptions of imperative features such as mutable state, first-class continuations, and exceptions has been an elusive goal for specification languages, which I believe is satisfactorily answered by the specifications in FORUM.

Although the extant formal presentations of pipelines specify the temporal behavior of the pipeline [TK93], they are unable to provide a concurrent computational specification of the pipelined operational semantics. The FORUM specification of the DLX pipeline is a concurrent executable logic program – one obtains a simulation tool for free. In my thesis, I specify the sequential and pipelined operational semantics for DLX, with important optimizations such as call-forwarding and early branch resolution [HP90]. Since floating-point operations and interrupts introduce unilluminating details to the specification, I have excluded them from the DLX instruction set that I specify. I prove the crucial *equivalence theorem* asserting the equivalence of the sequential and pipelined specifications of DLX. Furthermore, I define notions of program equivalences for DLX programs, and prove the correctness of the code rescheduling typically done for RISC machines [HP90]. The declarative specification of pipelined DLX operational semantics and the proofs of correctness of code rescheduling underlines the richness of FORUM as a specification language.

The point behind the specifications of imperative features is to extract the logical essence of imperative extensions of programming languages. The specification of state in natural semantics, higher order logic or other similar meta-theories [HMT89, MT92] represents state by non-logical means, such as a finite function, making the extension of state non-modular. In FORUM state is represented by logical propositions and is maintained in the sequent by logical rules. Thus, one can reason about state variables using cut-elimination. The understanding of this logical nature of imperative features underlines the richness of FORUM and its meta-theory.

The thesis has five main parts. In the first part I explain the logic programming methodology of FORUM, and then specify Higher-Order π Calculus ($HO\pi$-calculus), a typical calculus for concurrent processes, [MPW92a, MPW92b] substantiating the idea that FORUM provides an appropriate framework for specifying concurrent processes. In the second part, I specify UML in stages starting from the functional part of UML, and adding exceptions, mutable store and callcc modularly in separate steps. I prove that the FORUM specification of UML without callcc is the same as the specification in [HMT89]. In the third part, I define a notion of program equivalence induced by the translation into FORUM, and prove that it coincides with the standard definition of program equivalence. Furthermore, using FORUM specifications I prove several of the program equivalences involving mutable store in [MS88, Sie93, SF92, OT93]. In the fourth part, I specify the sequential and pipelined operational semantics for DLX. I also prove that the sequential and pipelined operational semantics are equivalent. Furthermore, I formulate a notion of program equivalence for DLX programs, and prove correctness of code-rescheduling for the DLX machine. In the final section, I explain the extensions that I seek of my current research and how I intend to carry those out.

Chapter 2

FORUM

In this chapter I introduce FORUM, a new meta-logic proposed in [Mil94]. FORUM can encode linear logic [Gir87] without using any non-logical constants. On the one hand, provability in FORUM is the same as provability in linear logic. On the other hand, in FORUM all right hand rules permute with each other — a property which is not true of the proof systems for linear logic in [Gir87]. The novelty of FORUM is in the choice of connectives which makes all right hand rules permute. Since uniform proofs are complete for FORUM, following [MNPS91] a logic programming language can be designed for FORUM. FORUM extends earlier work in designing logic programming languages from linear logic [HM91, AP90] in the sense that the provability in FORUM is the same as the provability in linear logic.

Clauses in FORUM can have multiple heads. I show some programming examples which exploit multiple heads to represent synchronization in FORUM. The basic intuition is that concurrent computations can be represented in FORUM. I substantiate this claim by translating a particular presentation of a fragment of Higher-Order π ($HO\pi$-calculus) calculus [San92a] which is rich enough to encode Lazy Lambda calculus [San92b, San92a, Chi94]. The parallel combinator of $HO\pi$-calculus is mapped to $\wp$, the multiplicative disjunction of Linear Logic. This identification of concurrency with proof search in a multiple conclusion logic where all the right-hand rules permute with each other makes the intuitions technically precise. The handling of names using the restriction opreator in $HO\pi$-calculus and the universal quantifier in FORUM are quite different, and consequently the translation into FORUM is sound but not complete. The computational mechanism of FORUM provides a new flavor of process theories which are very expressive, and the translation shows how computations maybe mapped from $HO\pi$-calculus to FORUM.

In the first section, I introduce FORUM and define its syntax and proof rules precisely. I illustrate the programming style in the presence of multiple heads via some examples. In the second section, I introduce the syntax, structural equivalence, and reduction semantics of $HO\pi$-calculus. I translate $HO\pi$-calculus into FORUM and prove that if process P reduces to Q, then the translation of Q entails the translation of P. Furthermore, I illustrate the difference between the restriction operator of $HO\pi$-calculus and the universal quantifier in FORUM.

2.1 FORUM — Logic programming with multiple heads

FORUM [Mil94], is best explained as a particular presentation for Linear Logic which gives us access to the entire Linear Logic as a logic programming language in the sense of [MNPS91, AP90]. Linear logic was introduced in [Gir87] as a new logic which decomposed the connectives of the familiar classical and intuitionistic logics. This finer analysis of connectives had immediate implications in the design of logic programming languages which analyzed connectives as directions for proof search [MNPS91]. LO[AP90] and Lolli [HM91] were two new logic programming languages which resulted from different sets of connectives of linear logic. However, the logical constants in neither of these languages were rich enough to encode the entire linear logic. In [Mil94] classical linear logic is encoded in FORUM without using any non-logical constants. The logical connectives in FORUM are $-\!\circ$, $\Rightarrow$, $\wp$, $\&$, $\forall$, $\perp$ and $\top$. $-\!\circ$, $\wp$, $\&$, $\forall$, $\perp$ and $\top$ are linear logic connectives as defined in [Gir87]. Instead of the modalities of linear logic, FORUM has $\Rightarrow$, the intuitionistic implication. In [Mil94] it was proved that the proof system of FORUM has the uniform proof and focussing property [MNPS91, AP90]. FORUM can be thus thought of Linear Logic and Church's simple theory of types [Chu40] put together. I begin by defining the syntax of FORUM.

Definition 2.1 [Types, Terms and Formulas in FORUM] Let Θ be a set of base types and $o \in \Theta$ the type of propositions. The set of well formed types is defined as

- if $\sigma \in \Theta$ then σ is a type, and
- if τ_1 and τ_2 are types then so is $\tau_1 \rightarrow \tau_2$.

Let Σ be a set of pairs whose first component is a term, and the second component is the type of the term, written as $f : \tau$, if f is a term with type τ in Σ. $\multimap : o \rightarrow o \rightarrow o$, $\Rightarrow : o \rightarrow o \rightarrow o$, $\wp : o \rightarrow o \rightarrow o$, $\& : o \rightarrow o \rightarrow o$, $\forall_\tau : (\tau \rightarrow o) \rightarrow o$, $\perp : o$ and $\top : o$ are the logical constants in Σ. $\Rightarrow$ denotes intuitionistic implication and the infix symbol $\circ\!-$ denotes the converse of $\multimap$. The set of terms over Σ is defined as :

- If $c : \tau \in \Sigma$ then c is a term of type τ.
- If $f : \tau \rightarrow \sigma$ and $t : \tau$ then $(f\, t)$ is a term of type σ.
- If x is a variable of type σ and $t : \tau$ then $\lambda x.\, t$ is of type $\sigma \rightarrow \tau$.

Terms of type o are defined to be formulas. The *order of a type* $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0$ is the one plus the max of the orders of $\tau_1 \ldots \tau_n$, and $\tau_0 \in \Theta$. The order of the elements of Θ is 0. For a non-logical constant $c : \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0 \in \Sigma$, τ_0 is a member of Θ, and if τ_0 is o then c is called a *predicate*.

I follow [Bar84] in conventions regarding free and bound variables and α conversion. As usual, $\rightarrow$ associates to the right and application to the left. The logical constants are written in the familiar infix form, and I write $\forall_\tau \lambda x.\, t$ as $\forall x : \tau.\, t$. Let $t = s$, for λ-terms t and s mean that t and s are α equivalent. If the variable x and term s are of the same type, then

$t[x := s]$ denotes the capture-free substitution of s for x in t. Besides α conversion, terms are also related by the following rules of β and η conversions:

- The term s_1 β-converts to the term s_2 if s_1 contains a subformula occurrence of the form $((\lambda x.\, t_1)\; t_2)$ and s_2 arises from replacing that subformula occurrence with $t_1[x := t_2]$.
- The term s_1 η-converts to the term s_2 if s_1 contains a subformula occurrence of the form $\lambda x.(t\, x)$, in which x is not free in t, and s_2 arises from replacing that subformula occurrence with t.

■

The proof system for FORUM as presented in figure 2.1 is a minor variation on the one in [Mil94]. The sequents comprise of five parts, Σ, Ψ, Δ, B and Γ. The signature of the terms in the sequent is given by Σ. The intuitionistic part of the sequent, Ψ, is treated like a set, *i.e.* contraction, weakening and exchange are allowed on formulas in Ψ. The linear parts of the sequent, Δ and Γ, are treated as multisets of formulas, allowing only exchange on these parts of the sequent. In the sequent $\Sigma : \Psi\, ; \Delta \xrightarrow{B} \Gamma$, one applies left rules to the formula B. By abuse of notation I write B, Ψ to mean $\{B\} \cup \Psi$, and Δ_1, Δ_2 to stand for the multiset union of the multisets Δ_1 and Δ_2, and B, Δ_2 to stand for the multiset union of the multisets $\{B\}$ and Δ_2. $\Psi[c := t]$ denotes the capture-free substitution of t for c in all formulas in the set Ψ and $\Delta[c := t]$ denotes the capture-free substitution of t for c in all formulas in the multiset Δ. I write $A \equiv_\Sigma B$ as an abbreviation for the statement that both $\Sigma :\, ; A \longrightarrow B$ and $\Sigma :\, ; B \longrightarrow A$ are provable in FORUM. I say that A is logically equivalent to B when $A \equiv_\Sigma B$. One can prove a cut-elimination theorem for FORUM, which states that *CutL, CutI, and CutS* are redundant in FORUM [Mil94]. Also note that if $\Sigma : \Psi\, ; \Delta \longrightarrow \Gamma$ is provable, $\Sigma \subset \Sigma_1$ and $\Psi \subset \Psi_1$, then $\Sigma_1 : \Psi_1\, ; \Delta \longrightarrow \Gamma$ is also provable.

Following the line of reasoning in [MNPS91], logic programs can be viewed as collections of formulas specifying the meaning of non-logical constants, and computation is identified

$$\frac{\Sigma : \Psi\, ;\, \Delta_1 \longrightarrow B, \Gamma_1 \quad \Sigma : \Psi\, ;\, \Delta_2, B \longrightarrow \Gamma_2}{\Sigma : \Psi\, ;\, \Delta_1, \Delta_2 \longrightarrow \Gamma_1, \Gamma_2}\ \mathit{CutL}$$

$$\frac{\Sigma : \Psi\, ;\, \longrightarrow B \quad \Sigma : \Psi, B\, ;\, \Delta \longrightarrow \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow \Gamma}\ \mathit{CutI}$$

$$\frac{t \text{ is a } \Sigma\text{-term of type } \tau \quad \Sigma, c : \tau : \Psi\, ;\, \Delta \longrightarrow \Gamma}{\Sigma : \Psi[c := t]\, ;\, \Delta[c := t] \longrightarrow \Gamma[c := t]}\ \mathit{CutS}$$

$$\frac{}{\Sigma : \Psi\, ;\, \Delta \longrightarrow \top, \Gamma}\ \top\text{-}R \qquad \frac{\Sigma : \Psi\, ;\, \Delta \longrightarrow \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow \bot, \Gamma}\ \bot\text{-}R$$

$$\frac{\Sigma : \Psi\, ;\, \Delta \longrightarrow B, \Gamma \quad \Sigma : \Psi\, ;\, \Delta \longrightarrow C, \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow B \,\&\, C, \Gamma}\ \&\text{-}R$$

$$\frac{\Sigma : \Psi\, ;\, B, \Delta \longrightarrow C, \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow B \multimap C, \Gamma}\ \multimap\text{-}R \qquad \frac{\Sigma : B, \Psi\, ;\, \Delta \longrightarrow C, \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow B \Rightarrow C, \Gamma}\ \Rightarrow\text{-}R$$

$$\frac{\Sigma : \Psi\, ;\, \Delta \longrightarrow B, C, \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow B \wp C, \Gamma}\ \wp\text{-}R \qquad \frac{y : \tau, \Sigma : \Psi\, ;\, \Delta \longrightarrow B[y/x], \Gamma}{\Sigma : \Psi\, ;\, \Delta \longrightarrow \forall x : \tau.\, B, \Gamma}\ \forall\text{-}R$$

$$\frac{\Sigma : \Psi\, ;\, \Delta \xrightarrow{B} \Gamma}{\Sigma : \Psi\, ;\, B, \Delta \longrightarrow \Gamma}\ \text{decide1} \qquad \frac{\Sigma : B, \Psi\, ;\, \Delta \xrightarrow{B} \Gamma}{\Sigma : B, \Psi\, ;\, \Delta \longrightarrow \Gamma}\ \text{decide2}$$

$$\frac{}{\Sigma : \Psi\, ;\, \xrightarrow{A} A}\ \mathit{initial} \qquad \frac{}{\Sigma : \Psi\, ;\, \xrightarrow{\bot}}\ \bot\text{-}L$$

$$\frac{\Sigma : \Psi\, ;\, \Delta \xrightarrow{B} \Gamma}{\Sigma : \Psi\, ;\, \Delta \xrightarrow{B \,\&\, C} \Gamma}\ \&\text{-}L \qquad \frac{\Sigma : \Psi\, ;\, \Delta \xrightarrow{C} \Gamma}{\Sigma : \Psi\, ;\, \Delta \xrightarrow{B \,\&\, C} \Gamma}\ \&\text{-}L$$

$$\frac{\Sigma : \Psi\, ;\, \Delta_1 \longrightarrow B, \Gamma_1 \quad \Sigma : \Psi\, ;\, \Delta_2 \xrightarrow{C} \Gamma_2}{\Sigma : \Psi\, ;\, \Delta_1, \Delta_2 \xrightarrow{B \multimap C} \Gamma_1, \Gamma_2}\ \multimap\text{-}L$$

$$\frac{\Sigma : \Psi\, ;\, \longrightarrow B \quad \Sigma : \Psi\, ;\, \Delta \xrightarrow{C} \Gamma}{\Sigma : \Psi\, ;\, \Delta \xrightarrow{B \Rightarrow C} \Gamma}\ \Rightarrow\text{-}L$$

$$\frac{\Sigma : \Psi\, ;\, \Delta_1 \xrightarrow{B} \Gamma_1 \quad \Sigma : \Psi\, ;\, \Delta_2 \xrightarrow{C} \Gamma_2}{\Sigma : \Psi\, ;\, \Delta_1, \Delta_2 \xrightarrow{B \wp C} \Gamma_1, \Gamma_2}\ \wp\text{-}L$$

$$\frac{t \text{ is a } \Sigma\text{-term of type } \tau \quad \Sigma : \Psi\, ;\, \Delta \xrightarrow{B[t/x]} \Gamma}{\Sigma : \Psi\, ;\, \Delta \xrightarrow{\forall x : \tau. B} \Gamma}\ \forall\text{-}L$$

Figure 2.1: The rule $\forall$-R has the proviso that y is not declared in the signature Σ.

$$
\begin{array}{rcl}
(\mathsf{neq}\ (\mathsf{s}\ X)\ \mathsf{z}) & \circ\!- & \mathbf{1} \\
(\mathsf{neq}\ \mathsf{z}\ (\mathsf{s}\ X)) & \circ\!- & \mathbf{1} \\
(\mathsf{neq}\ (\mathsf{s}\ X)\ (\mathsf{s}\ Y)) & \circ\!- & (\mathsf{neq}\ X\ Y)
\end{array}
$$

Figure 2.2: Specifying inequality of natural numbers in FORUM

with the search for *uniform proofs.* The key feature of FORUM is that the right-hand side of the sequent can now have more than one formula. *How does one interpret uniform proofs in the presence of more than one goal formula ?* The answer in [Mil94] comes from the concept of permutabilities in proof theory [Kle64]. Informally, it is required that the order in which goal formulas are processed does not affect the success of the proof search. This novelty lets us represent concurrent computations in FORUM, as was exhibited in [Mil94] by specifying Algol-like implementations of CML, and First-order π-Calculus in [Mil93].

I illustrate the computational mechanism of FORUM with a simple example. I begin by specifying the natural numbers in FORUM. I introduce a new type in FORUM called nat and two new constants $\mathsf{z} : \mathsf{nat}$ and a function $\mathsf{s} : \mathsf{nat} \rightarrow \mathsf{nat}$. The intended meaning is that nat is the type of natural numbers, z denotes 0, and s denotes the successor function. I now want to define a predicate $\mathsf{neq} : \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow o$, which should be provable of the two terms m and n of type nat, if m is not equal to n. The meaning of neq is specified by the universal closure of the clauses in figure 2.2, called C_{neq}. The collection of clauses is called the program for the non-logical constant neq. In the specification I use a new logical connective $\mathbf{1}$. It can be defined as $\perp \multimap \perp$, and the proof rule can be derived correspondingly. I show the right-hand side rule below.

$$\frac{}{\Sigma : \Psi\ ;\ \longrightarrow\ \mathbf{1}}\ \mathbf{1}-R$$

To check neq for $(\mathsf{s}\ \mathsf{z})$ and $(\mathsf{s}\ \mathsf{s}\ \mathsf{z})$, I try to construct a proof of

$$\mathsf{z} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \rightarrow \mathsf{nat} : C_{neq}\ ;\ \longrightarrow\ (\mathsf{neq}\ (\mathsf{s}\ \mathsf{z})\ (\mathsf{s}\ \mathsf{s}\ \mathsf{z})).$$

If the sequent is provable then the two numbers are not equal. For the example at hand, the proof is constructed below. The first rule I apply is the *backchain* rule. The backchain rule is an abbreviation, instead of constructing the following proof

$$\dfrac{\dfrac{\dfrac{\Sigma : \Psi, C \multimap B ; \Delta \overset{\delta}{\longrightarrow} C, \Gamma \qquad \dfrac{}{\Sigma : \Psi, C \multimap B ; \Delta \xrightarrow{B} B}\ initial}{\Sigma : \Psi, C \multimap B ; \Delta \xrightarrow{C \multimap B} B, \Gamma}\ \multimap -L}{\Sigma : \Psi, C \multimap (B_1 \wp .. \wp B_n) ; \Delta \longrightarrow (B_1 \wp .. \wp B_n), \Gamma_1, \Gamma_2, \ldots, \Gamma_n, \Gamma_{n+1}}\ decide2}{\Sigma : \Psi, C \multimap (B_1 \wp \ldots \wp B_n) ; \Delta \longrightarrow \Gamma_1, B_1, \Gamma_2, \ldots, \Gamma_n, B_n, \Gamma_{n+1}}\ exchange, \wp - R$$

I abbreviate it as

$$\dfrac{\Sigma : \Psi, C \multimap (B_1 \wp \ldots \wp B_n) ; \Delta \overset{\delta}{\longrightarrow} C, \Gamma_1, \Gamma_2, \ldots, \Gamma_n, \Gamma_{n+1}}{\Sigma : \Psi, C \multimap (B_1 \wp \ldots \wp B_n) ; \Delta \longrightarrow \Gamma_1, B_1, \Gamma_2, \ldots, \Gamma_n, B_n, \Gamma_{n+1}}\ backchain$$

B is an abbreviation for $B_1 \wp \ldots \wp B_n$, and Γ is an abbreviation for $\Gamma_1, \ldots, \Gamma_{n+1}$.

The formula on the right hand side unifies the head of the clause $(\mathsf{neq}\ (\mathsf{s}\ X)\ (\mathsf{s}\ Y)) \multimapinv (\mathsf{neq}\ X\ Y)$. So I then have to prove $(\mathsf{neq}\ \mathsf{z}\ (\mathsf{s}\ \mathsf{z}))$. The goal now unifies with the head of the clause $(\mathsf{neq}\ \mathsf{z}\ (\mathsf{s}\ X)) \multimapinv \mathbf{1}$, leaving me to prove $\mathbf{1}$. I complete the proof using the $\mathbf{1}$ rule. In this manner, proof search for cut-free proofs is identified with computation.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{z} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \to \mathsf{nat} : C_{neq} ;\ \longrightarrow \mathbf{1}}\ \mathbf{1}}{\mathsf{z} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \to \mathsf{nat} : C_{neq} ;\ \longrightarrow (\mathsf{neq}\ \mathsf{z}\ (\mathsf{s}\ \mathsf{z}))}\ backchain}{\mathsf{z} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \to \mathsf{nat} : C_{neq} ;\ \longrightarrow (\mathsf{neq}\ (\mathsf{s}\ \mathsf{z})\ (\mathsf{s}\ \mathsf{s}\ \mathsf{z}))}\ backchain$$

The novelty of FORUM lies in the fact that the clauses can contain multiple heads, *i.e.* formulas like $A \wp B$. I want to specify a predicate inc: $(\mathsf{nat} \to o) \to o$, which takes a predicate of type $\mathsf{nat} \to o$ as an argument, such that every time inc is executed in a proof a new number is returned. Let ctr: $\mathsf{nat} \to o$ be a non-logical constant, denoting a memory cell in the environment storing the next number to be used by inc. Suppose ctr is initialized to some number, then the only clause required will be the universal closure of the following clause called *Inc*.

$$[(\mathsf{inc}\ P) \wp (\mathsf{ctr}\ X)] \multimapinv [(P\ X) \wp (\mathsf{ctr}\ (\mathsf{s}\ X))].$$

Both ($\mathsf{inc}\ P$) and ($\mathsf{ctr}\ X$) must be on the right hand side of the sequent before one can backchain on the clause. *In this sense, clauses with multiple heads enforce synchronization between various predicates.* The last step in the uniform proof of

$$\mathsf{inc} : (\mathsf{nat} \to o) \to o, \mathsf{ctr} : \mathsf{nat} \to o, P : \mathsf{nat} \to o : \mathit{Inc} \,;\ \longrightarrow\ (\mathsf{inc}\ P) \wp (\mathsf{ctr}\ (\mathsf{s}\ \mathsf{z}))$$

is shown below.

$$\dfrac{\begin{array}{c}\vdots \\ \mathsf{ctr}, \mathsf{inc}, P : \mathit{Inc} \,;\ \longrightarrow\ (P\ (\mathsf{s}\ \mathsf{z})) \wp (\mathsf{ctr}\ (\mathsf{s}\ \mathsf{s}\ \mathsf{z}))\end{array}}{\mathsf{ctr}, \mathsf{inc}, P : \mathit{Inc} \,;\ \longrightarrow\ (\mathsf{inc}\ P) \wp (\mathsf{ctr}\ (\mathsf{s}\ \mathsf{z}))}\ \mathit{backchain}$$

To use the clause *Inc*, I need to have both ($\mathsf{inc}\ P$) and ($\mathsf{ctr}\ (\mathsf{s}\ \mathsf{z})$), or, in other words, ($\mathsf{inc}\ P$) and ($\mathsf{ctr}\ (\mathsf{s}\ \mathsf{z})$) synchronize with each other. Now, by backchaining on *Inc* the goal becomes $(P\ (\mathsf{s}\ \mathsf{z})) \wp (\mathsf{ctr}\ (\mathsf{s}\ \mathsf{s}\ \mathsf{z}))$. Note that because ctr is linear it is destructively read by the *Inc* clause — the number stored in ctr is increased by one as a result of backchaining.

The examples above point towards a relationship between concurrency and proof search in multiple conclusion logic where the right hand side rules permute. The idea is that all the processes on the right hand side are free to compute concurrently and synchronize with each other. Some of these intuitions will be made precise in the next section, where I specify a particular presentation of a fragment of $HO\pi$-calculus in FORUM.

2.2 Specifying $HO\pi$ in FORUM

In this section I specify a fragment of $HO\pi$-calculus as defined in [San92a] to make concrete my claim that FORUM can be used to represent both abstraction and concurrency. I begin with a brief presentation of $HO\pi$-calculus, and refer the interested reader to [Mil89, MPW92a, MPW92b, San92b, San92a] for a detailed introduction to, and analyses of $HO\pi$-calculus and π-calculus. The motivating idea of $HO\pi$-calculus is to provide higher order communication in the framework of synchronous mobile process algebras. The fragment of $HO\pi$-calculus that I consider can encode call-by-value and call-by-name lambda calculus [Mil90, San92b, San92a]. I use $x, y, \ldots$ possibly subscripted, to stand for *Names* and their capitalized versions to range over *Vars*. Moreover, K stands for a process or a name, and U stands for a variable or a name.

Definition 2.2 [Syntax for $HO\pi$-calculus.] The processes in $HO\pi$-calculus are defined by P and the prefixes by α.

$$
\begin{array}{lll}
P & ::= & \mathsf{0} \\
 & & \mid (X) \\
 & & \mid (P \mid P) \\
 & & \mid (\nu x(P)) \\
 & & \mid (\alpha.P) \\
 & & \\
\alpha & ::= & x(U) \mid \overline{x}\langle K \rangle
\end{array}
$$

■

$\mathsf{0}$ is the inactive process — not capable of any action or interaction. $x(U).P$ accepts input for variable U along the channel x, while $\overline{x}\langle K \rangle.P$ transmits K along the channel x. $\nu x(P)$ makes the name x private to the process P, and $P \mid Q$ places the two processes P and Q in parallel. The variable U is bound in $x(U).P$, and x is bound in $\nu x(P)$. I often abbreviate $\alpha.\mathsf{0}$ as α. The reduction relation for $HO\pi$-calculus is divided in two parts, the structural equivalence and the reduction relation. We illustrate the reduction semantics and explain the syntax conceptually with some examples.

Example 2.3 Examples of reductions in $HO\pi$-calculus.

1. $\overline{x}\langle Q \rangle.P \mid x(Y).R \rightarrow_\pi P \mid R[Y := Q]$
2. $\nu x(\overline{w}\langle x \rangle.P \mid Q) \mid w(y).R \rightarrow_\pi \nu x(Q \mid R[y := x]) \mid P$, $x \notin FV(P)$ and $x \notin FV(w(y).R)$.
3. $\overline{x}\langle Q \rangle.\mathsf{0} \mid x(Y).(Y \mid P) \rightarrow_\pi Q \mid P$

$$\frac{}{\overline{x}\langle Q\rangle.P \mid x(Y).R \;\Rightarrow_\pi\; P \mid R[Y := Q]}\;ComP$$

$$\frac{}{\overline{x}\langle y\rangle.P \mid x(z).R \;\Rightarrow_\pi\; P \mid R[z := y]}\;ComN$$

$$\frac{P \;\Rightarrow_\pi\; Q}{\nu x(P) \;\Rightarrow_\pi\; \nu x(Q)}\;\nu R$$

$$\frac{P \;\Rightarrow_\pi\; P_1}{P \mid Q \;\Rightarrow_\pi\; P_1 \mid Q}\;Par$$

$$\frac{P \;\equiv_\pi\; P_1 \quad P_1 \;\Rightarrow_\pi\; Q_1 \quad Q_1 \;\equiv_\pi\; Q}{P \;\Rightarrow_\pi\; Q}\;Struct$$

$$\frac{}{P \;\Rightarrow_\pi\; P}\;Refl$$

$$\frac{P \;\Rightarrow_\pi\; Q_1 \quad Q_1 \;\Rightarrow_\pi\; Q}{P \;\Rightarrow_\pi\; Q}\;Trans$$

Figure 2.3: Reduction semantics for $HO\pi$-calculus

■

In example 2.3.1, $\overline{x}\langle Q\rangle.P$ transmits Q along the channel x and $x(Y).R$ receives Q on the x. The rule underlines one key feature of the calculus — *the communications are synchronous*, *i.e.* a process, *e.g.* $\overline{x}\langle Q\rangle.P$, that wants to send a message waits until there is a process, *e.g.* $x(Y).P$, in the environment which will accept that message, hence the name *synchronous message passing.* This also ensures a flavor of sequencing in the process, *e.g.* in $\alpha_1.\alpha_2.P$ the action corresponding to α_1 must happen before the action for α_2 can occur. Another basic feature of $HO\pi$-calculus is the capability of changing the connectivity amongst processes during computation. In example 2.3.2, the channel x in $\nu x(\overline{w}\langle x\rangle.P \mid Q)$ is a private channel between $\overline{w}\langle x\rangle.P$ and Q. However, it is possible for x to *extrude its scope* and be sent to $w(y).R$. In this sense, the connectivity of the processes can change during the computation, and hence the name *mobile processes.* Example 2.3.3 illustrates the novelty of Higher- order processes. $x(Y).(Y \mid P)$ can be viewed as a process that will execute whatever process it receives from the environment in parallel with P. Higher-order communications provides the familiar substitution of λ-calculus in the context of $HO\pi$-calculus. [San92b, San92a] gave a nice encoding of Lazy λ-calculus in $HO\pi$-calculus and showed the correspondence between $HO\pi$-calculus and First Order π. [Chi94] compares the encoding of Lazy λ-calculus in $HO\pi$-calculus to the Continuation-passing semantics. Now, I define the structural equivalence and reduction semantics of $HO\pi$-calculus formally.

Definition 2.4 [Structural Equivalence, $\equiv_\pi$] $\equiv_\pi$ is defined as the least congruence containing the following rules.

1. $P \mid \mathsf{0} \;\equiv_\pi\; P$.
2. $P \mid Q \;\equiv_\pi\; Q \mid P$.
3. $(P \mid Q) \mid R \;\equiv_\pi\; P \mid (Q \mid R)$.
4. $\nu x(P) \mid Q \;\equiv_\pi\; \nu x(P \mid Q)$, $x \notin FV(Q)$.
5. $\nu x(P) \;\equiv_\pi\; P$, $x \notin FV(P)$.
6. $\nu x(\nu y(P)) \;\equiv_\pi\; \nu y(\nu x(P))$.

■

The reduction semantics for $HO\pi$-calculus is specified as an unlabeled system in terms of proof rules. One notable feature is that the separation of structural equivalence from reduction rules enables a concise presentation of the latter. It should be noted that reduction is built as a congruence for all term constructors *except prefixing*, as seen in the rules νR, *Par*, *agentR* and *Struct*; this feature forces a strict order of evaluation on prefixed processes, as we saw in example 2.3.1.

Definition 2.5 [Translation of $HO\pi$-calculus in FORUM] Let i be a new basic type, the type of names, and Σ_π be the set consisting of the following constants needed to describe the translation of $HO\pi$-calculus into FORUM.

$$
\begin{array}{rcl}
\mathsf{send}_{nm} & : & i \rightarrow i \rightarrow o \rightarrow o \\
\mathsf{receive}_{nm} & : & i \rightarrow (i \rightarrow o) \rightarrow o \\
\mathsf{send}_{pr} & : & i \rightarrow o \rightarrow o \rightarrow o \\
\mathsf{receive}_{pr} & : & i \rightarrow (o \rightarrow o) \rightarrow o
\end{array}
$$

The translation $()^o$ takes a process in $HO\pi$-calculus to a formula, $i.e.$ terms of type o, in FORUM.

$$
\begin{array}{rcl}
\mathsf{0}^o & = & \bot \\
X^o & = & X \\
(P \mid Q)^o & = & P^o \wp Q^o \\
(\nu x(P))^o & = & \forall x : i.\, P^o \\
(\overline{x}\langle Q\rangle.P)^o & = & \mathsf{send}_{pr}\ x\ Q^o\ P^o \\
(x(Y).P)^o & = & \mathsf{receive}_{pr}\ x\ \lambda Y.\, P^o \\
(\overline{x}\langle y\rangle.P)^o & = & \mathsf{send}_{nm}\ x\ y\ P^o \\
(x(z).P)^o & = & \mathsf{receive}_{nm}\ x\ \lambda z.\, P^o
\end{array}
$$

Let $\mathcal{E}_\pi$ be the set consisting of the universal closure of the following clause, which describes the meaning of the non-logicals in Σ_π.

$$
\begin{array}{rcll}
(\mathsf{receive}_{nm}\ x\ R) \wp (\mathsf{send}_{nm}\ x\ y\ P) & \multimapinv & (R\, y) \wp P, & \mathit{NameCl} \\
(\mathsf{receive}_{pr}\ x\ R) \wp (\mathsf{send}_{pr}\ x\ Q\ P) & \multimapinv & (R\, Q) \wp P, & \mathit{ProcessCl}
\end{array}
$$

■

Synchronization for processes exchanging names, $i.e.$ terms of type i, is specified by *NameCl*, while *ProcessCl* specifies synchronization for processes exchanging processes, $i.e.$ terms of type o. For the clauses in $\mathcal{E}_\pi$ to make sense, I need to show that the translation commutes with substitution. The proof is a straightforward induction on the structure of the process P.

Lemma 2.6 (Substitution Lemma for $()^o$) *Let P and Q be processes, then*

- $P^o[x := y] \;=\; P[x := y]^o$
- $P^o[X := Q^o] \;=\; P[x := Q]^o$

This translation is a simple extension of the one in [Mil93] to the case of $HO\pi$-calculus. Although I cover a small fragment of the calculus here, it is easy to extend these ideas along the line of reasoning in [Mil93] to handle richer versions of $HO\pi$-calculus. However, specifying $HO\pi$-calculus augmented with constants and agents, *i.e.* abstractions over variables or names in processes is rather subtle. The translation is fairly simple, and gives concrete intuition about the nature of concurrency in FORUM. Concurrency is identified with proof search in a multiple conclusion logic, where all the right-hand rules permute with each other. The usage of $\wp$ as the translation of the parallel combinator of $HO\pi$-calculus in the above translation makes this intuition technically precise. The fact that the intention of the $HO\pi$-calculus is captured by the translation is underlined by lemma 2.7 and theorem 2.8. Lemma 2.7 states that if two processes are structurally equivalent, then their translations are logically equivalent. The proof for lemma 2.7 follows from the logical equivalences in [Gir87]. Let $P^* \;=_{def}\; \forall \vec{x}.\, P^o$, where $\vec{x}$ are all variables of type i in the signature of P^o. Theorem 2.8 states that if P reduces to Q, then Q^* entails P^*. In the proof of theorem 2.8, going from left to right, the proof for theorem 2.8 is an induction on the height of reduction in $HO\pi$-calculus.

Lemma 2.7 *Given two processes P and Q in $HO\pi$-calculus, if $P \equiv_\pi Q$ then $P^o \equiv_{\Sigma_\pi} Q^o$.*

Theorem 2.8 *Let P and Q be $HO\pi$-calculus processes, and Σ contain the names and process variables in P and Q.*

$$\textit{if } P \Rightarrow_\pi Q \textit{ then } \Sigma_\pi, \Sigma : \mathcal{E}_\pi \,;\, Q^* \longrightarrow P^*$$

is provable in FORUM.

The reductions in translated processes have a simple shape. One essentially reorganizes the shape of the process using structural equivalences until either *NameCl* or *ProcessCl* is applicable. Although it is an interesting topic, I do not consider the issue of explaining interesting equivalence relations on $HO\pi$-calculus in FORUM, because the above translation makes my point — *concurrent computations can be represented in FORUM.* In [Mil93] bisimulations and trace equivalences for First Order π-calculus without ν are analyzed in the framework of a FORUM like language.

The translation of $HO\pi$-calculus into FORUM is proved sound by theorem 2.8, but is it complete ? This question was answered affirmatively in [Mil93] for the π-calculus without the restriction operator. However, in FORUM

$$\Sigma : ; \forall x.\, \forall y.\, Q[c := x][d := y] \;\longrightarrow\; \forall z.\, Q[c := z][d := z]$$

is provable, where Q is the translation of some process into FORUM. If the translation is complete, then for any process Q, this would imply that

$$\nu x(Q) \;\Rightarrow_\pi\; \nu x(\nu y(Q)).$$

This is false. Hence, the universal quantifier has a logical nature which is richer than that of the restriction operator in $HO\pi$-calculus. In the later chapters, I explore various specifications in the "process theory" obtained from FORUM, and its expressive power and abstraction mechanisms are made clear. Nonetheless, the question of a proof theoretic analogue of $HO\pi$-calculus remains, and there is some work in progress with new quantifiers which might shed more light on the restriction operator.

Chapter 3

Specifying UML

In this chapter I specify UML (Untyped ML) — untyped core SML excluding pattern matching and data-types, augmented with callcc — in FORUM. I prove that a program P in UML without callcc evaluates to a value V as per the specification in FORUM, if and only if P evaluates to V as per the specification in [HMT89]. The *correspondence between* the two operational semantics is restricted to core UML without callcc because [HMT89] does not provide a specification for callcc. The main point of this section is that FORUM allows us to specify imperative features — exceptions, mutable state, first-class continuations — in a modular and declarative way. In particular, I first specify the functional core of UML, and then specify exceptions, mutable state, and first-class continuations independently. If I need the specification for the functional core, and any combination of the imperative parts of UML, I just put the corresponding specifications together. For example, if I want the specification for the functional core of UML with exceptions, all I have to do is put my specification for the functional core of UML together with the specification for exceptions!

Modularity of specifications is as helpful in understanding the design of a programming language as it is in the design of the language itself. However, modularity is crucial not only for such esoteric purposes as 'understanding' and 'designing' languages, but also for many practical concerns including proving correctness of implementations [HM92], verifying

$$
\begin{array}{rll}
M ::= & V & (\Lambda_v) \\
 & \mid (f\ M\ M) & f \in \mathcal{O} \\
 & \mid (M\ M) & \\
 & \mid (\mathsf{if}\ M\ M\ M) & \\
 & \mid (\mathsf{let\ val}\ x = M\ \mathsf{in}\ M) & \\
 & \mid (\mathsf{let\ fun}\ f\ x = M\ \mathsf{in}\ M) & \\
 & & \\
V ::= & x & x \in \mathit{Vars}, (\mathit{Values}_v) \\
 & \mid n & n \in \mathcal{Z} \\
 & \mid b & b \in \mathcal{B} \\
 & \mid (\lambda x.\, M) & \\
 & \mid \bullet & \\
 & & \\
E ::= & [\,] & (\mathit{EvCont}_v) \\
 & \mid E[f\ [\,]\ M] & f \in \mathcal{O} \\
 & \mid E[f\ V\ [\,]] & f \in \mathcal{O} \\
 & \mid E[[\,]\ M] & \\
 & \mid E[V\ [\,]] & \\
 & \mid E[\mathsf{if}\ [\,]\ M\ M] & \\
 & \mid E[\mathsf{let\ val}\ x = [\,]\ \mathsf{in}\ M] &
\end{array}
$$

Figure 3.1: Syntax for Λ_v

correctness of optimizations in compilers, understanding program equivalences, and proving meta-theoretic properties such as type soundness in statically typed languages [WF91].

The plan for this chapter is to introduce the separate parts of UML in stages, and provide their specifications. I first introduce the syntax and operational semantics for Λ_v, the functional core of UML, which is most familiar and similar to the language considered in [HM92, MP92]. Next, I specify Λ_v in FORUM, and define when a translated program in FORUM evaluates to a value. I then prove the *correspondence theorem* between the FORUM specification and the specification in [HMT89]. The program is extended modularly to the functional core with exceptions, state, and continuations.

3.1 Λ_v — Functional Core of UML

The syntax of Λ_v - the functional core of UML - is very similar to the functional part of

SML without data-types. The syntax of Λ_v is defined formally in figure 3.1. The language contains integers, $\mathcal{Z}$, and booleans, $\mathcal{B}$, as constants. Arithmetic operators and equality test for integers are included in the language as term constructors in the set $\mathcal{O}$. Functional abstraction in the form of λ abstraction and application are represented by $\lambda x.\, M$ and $(M\ N)$ respectively. I include in the language $\mathsf{let\ val}\ x = M\ \mathsf{in}\ N$ which is treated like $((\lambda x.\, N)\, M)$. $\mathsf{let\ fun}\ f\, x = M\ \mathsf{in}\ N$ allows recursive definitions in Λ_v. $\bullet$ is a token, like the only value of type unit in SML [HMT89]. The evaluation contexts, $EvCont_v$ in figure 3.1, are a way of parsing a given Λ_v-term to find out the next redex to be contracted during evaluation. One can write standard functional programs in this language. The following program, exp, calculates m raised to the power n for non-negative numbers m and n. The constant $-$ stands for subtraction and $*$ for multiplication.

$$
\begin{array}{l}
\mathsf{let\ fun}\ f\ x = \lambda y, z.\, \mathsf{if}\ (= x\, 1)\ y\ (f\, (-x\, 1)\, (*y\, z)\, z) \\
\quad \mathsf{in} \\
\quad \lambda y, x.\, \mathsf{if}\ (= x\, 0)\ 1\ (f\, x\, y\, y)
\end{array}
$$

The operational semantics for Λ_v as presented in figure 3.2 is culled out from the specification for SML in [HMT89] by using substitutions instead of environments. This style of presentation is called *natural semantics* following [Kah87]. The evaluator is presented as a series of rules, all of which have a simple format — to evaluate an expression first evaluate the subexpressions and then put the results together as per the outermost term constructor. For example, to evaluate $(M\ N)$, first evaluate M to a function, $\lambda x.\, P$, and N to a value U, then finally evaluate $P[x := U]$ to a value V — the value of $(M\ N)$. The evaluator thus specified is *call-by-value*, as the argument to a function is evaluated before it is passed to the function. $\underline{f\,V\,U}$ denotes the constant in Λ_v, which is obtained as a result of performing f viewed as an arithmetic operator on the numbers denoted by V and U, *e.g.* $\underline{+\,5\,4}$ denotes the constant 9.

The order of evaluation of the subterms is not a part of the syntax — it is an additional requirement that the hypothesis to the rules should be read from left-to-right. In a language with partial arithmetic operations such as $\div$, division, the order of evaluation is crucial.

$$\frac{}{c \Downarrow c}\; c \in \mathcal{Z} \cup \mathcal{B}$$

$$\frac{}{\lambda x.\, M \Downarrow \lambda x.\, M}$$

$$\frac{M \Downarrow \lambda x.\, P \quad N \Downarrow U \quad P[x := U] \Downarrow V}{M\ N \Downarrow V}$$

$$\frac{M \Downarrow U \quad N[x := U] \Downarrow V}{\mathsf{let\ val}\ x = M\ \mathsf{in}\ N \Downarrow V}$$

$$\frac{N[f := \lambda x.\, \mathsf{let\ fun}\ f\,x = M\ \mathsf{in}\ N] \Downarrow V}{\mathsf{let\ fun}\ f\,x = M\ \mathsf{in}\ N \Downarrow V}$$

$$\frac{M \Downarrow V \quad N \Downarrow U}{f\ M\ N \Downarrow \underline{f\,V\,U}}\; f \in \mathcal{O}$$

$$\frac{M \Downarrow \mathsf{true} \quad N \Downarrow V}{\mathsf{if}\ M\ N\ P \Downarrow V}$$

$$\frac{M \Downarrow \mathsf{false} \quad P \Downarrow V}{\mathsf{if}\ M\ N\ P \Downarrow V}$$

Figure 3.2: Natural Semantics specification for Λ_v

For example, consider the program $(+\,P\,(\div\,5\,0))$, where P is any non-terminating program. In a language with exceptions, evaluating the program left-to-right will cause it to diverge. However, evaluating the program right to left may raise a division-by-0 exception. Using the given specification, left to right evaluation will result in an infinite search for a computation tree reflecting the non-termination of P, whereas right-to-left evaluation will cause a finite evaluation tree in which one can detect the division-by-0 error.

The specification of Λ_v in FORUM requires that one translates Λ_v terms into the higher order abstract syntax of FORUM. The translation reveals the binding structure of the language. Issues such as capture free substitution and α-conversion in the object language, *i.e.* Λ_v, are taken care of by substitution and the binding mechanism of the meta-language, *i.e.* FORUM. The terms and types required to define the translation comprise the set Σ_v, and are defined in figure 3.3. I define two basic types, vl and tm, along with a coercion function, $\langle . \rangle$, mapping terms of type vl to terms of type tm. vl is the type of *values*, and tm is the type of *terms*. I then introduce terms at appropriate types to encode the terms of Λ_v,

$$
\begin{array}{rcll}
\mathsf{abs} & : & (\mathsf{vl} \rightarrow \mathsf{tm}) \rightarrow \mathsf{vl} & \\
c & : & \mathsf{vl} & c \in \mathcal{B} \cup \mathcal{Z} \cup \{\bullet\} \\
 & & & \\
\langle . \rangle & : & \mathsf{vl} \rightarrow \mathsf{tm} & \\
f & : & \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} & f \in \mathcal{O} \\
\mathsf{app} & : & \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} & \\
\mathsf{cond} & : & \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} & \\
\mathsf{letval} & : & (\mathsf{vl} \rightarrow \mathsf{tm}) \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} & \\
\mathsf{letfun} & : & (\mathsf{vl} \rightarrow \mathsf{tm}) \rightarrow (\mathsf{vl} \rightarrow \mathsf{vl} \rightarrow \mathsf{tm}) \rightarrow \mathsf{tm} & \\
\mathsf{ifbr} & : & \mathsf{vl} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} & \\
\mathsf{apply} & : & \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow o & \\
 & & & \\
\mathsf{eval} & : & \mathsf{tm} \rightarrow (\mathsf{vl} \rightarrow o) \rightarrow o &
\end{array}
$$

Figure 3.3: Σ_v, Signature for Λ_v evaluator

e.g. app is a term construct which will be the target of application terms in the translation. For every integer and boolean, I introduce a constant of vl type. For every operator f, I introduce a constant f in the signature of type $\mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm}$.

Following is the translation of exp into a term of type tm in FORUM.

$$
\begin{array}{l}
\mathsf{letfun}\ \lambda f.\,(\mathsf{abs}\ \lambda y.\,(\mathsf{abs}\ \lambda x.\,\mathsf{cond}\ (=x\ \langle 0\rangle)\ \langle 1\rangle\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ f\ x)\ y)\ y)))\\
\qquad \lambda f, x.\,(\mathsf{abs}\ \lambda y.\,(\mathsf{abs}\ \lambda z.\,\mathsf{cond}\ (=x\ \langle 1\rangle)\ y\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ f\ (-\ x\ 1))\ (*\ y\ z))\ z)))
\end{array}
$$

The first argument of letfun is the body of the $\mathsf{let\ fun}$ declaration, and the second argument is the function declaration. The body is parameterized over the function variable defined by the $\mathsf{let\ fun}$ as made explicit by the meta-level λ-binding of f in the first argument of letfun. The example also illustrates how the λ bindings in the object language get converted into λ-bindings in the meta-language. For example, $\lambda x.\,M$ is translated as $(\mathsf{abs}\ \lambda x.\,M^o)$, M^o is the translation of M, and the λ binding in abs is at the meta-level. Other than this clean explanation of variable bindings, the translation, although heavy on usage of new syntax, is similar in spirit to parsing concrete terms into abstract syntax trees. The translation of Λ_v terms to FORUM is rather cumbersome, but it is crucial to the statement of the *correspondence theorem* between FORUM specification and the natural semantics

$$
\begin{array}{rcll}
\phi_v(x) & = & x & \\
\phi_v(\lambda x.\, M) & = & \mathsf{abs}\ \lambda x : \mathsf{vl}.\ \mathcal{H}_v(M) & \\
\phi_v(c) & = & c & c \in \mathcal{Z} \cup \mathcal{B} \cup \{\bullet\} \\
 & & & \\
\mathcal{H}_v(V) & = & \langle \phi_v(V) \rangle & \\
\mathcal{H}_v(f\ M\ N) & = & f\ \mathcal{H}_v(M)\ \mathcal{H}_v(N) & \\
\mathcal{H}_v(M\ N) & = & \mathsf{app}\ \mathcal{H}_v(M)\ \mathcal{H}_v(N) & \\
\mathcal{H}_v(\mathsf{if}\ M\ N\ P) & = & \mathsf{cond}\ \mathcal{H}_v(M)\ \mathcal{H}_v(N)\ \mathcal{H}_v(P) & \\
\mathcal{H}_v(\mathsf{let\ val}\ x = M\ \mathsf{in}\ N) & = & \mathsf{letval}\ (\lambda x.\, \mathcal{H}_v(N))\ \mathcal{H}_v(M) & \\
\mathcal{H}_v(\mathsf{let\ fun}\ f\, x = M\ \mathsf{in}\ N) & = & \mathsf{letfun}\ (\lambda f.\, \mathcal{H}_v(N))\ (\lambda f, x.\, \mathcal{H}_v(M)) &
\end{array}
$$

Figure 3.4: $\phi_v : \mathit{Values}_v \to \mathsf{vl}$, $\mathcal{H}_v : \Lambda_v \to \mathsf{tm}$

$$
\begin{array}{rcll}
\psi_v(x) & = & x & \\
\psi_v(\mathsf{abs}\ \lambda x.\, M) & = & \lambda x.\, \mathcal{L}_v(M) & \\
\psi_v(c) & = & c & c \in \mathcal{Z} \cup \mathcal{B} \cup \{\bullet\} \\
 & & & \\
\mathcal{L}_v(\langle V \rangle) & = & \psi_v(V) & \\
\mathcal{L}_v(f\ M\ N) & = & f\ \mathcal{L}_v(M)\ \mathcal{L}_v(N) & f \in \mathcal{O} \\
\mathcal{L}_v(\mathsf{app}\ M\ N) & = & \mathcal{L}_v(M)\ \mathcal{L}_v(N) & \\
\mathcal{L}_v((\mathsf{ifbr}\ V\ N\ P)) & = & \mathsf{if}\ \psi_v(V)\ \mathcal{L}_v(N)\ \mathcal{L}_v(P) & b \in \mathcal{B} \\
\mathcal{L}_v(\mathsf{cond}\ M\ N\ P) & = & \mathsf{if}\ \mathcal{L}_v(M)\ \mathcal{L}_v(N)\ \mathcal{L}_v(P) & \\
\mathcal{L}_v(\mathsf{letval}\ R\ N) & = & \mathsf{let\ val}\ x = \mathcal{L}_v(N)\ \mathsf{in}\ \mathcal{L}_v(R\, x) & x\ \mathit{fresh} \\
\mathcal{L}_v(\mathsf{letfun}\ R_1\ R_2) & = & \mathsf{let\ fun}\ f\, x = \mathcal{L}_v(R_2\, f\, x)\ \mathsf{in}\ \mathcal{L}_v(R_1\, f) & f, x\ \mathit{fresh}
\end{array}
$$

Figure 3.5: $\psi_v : \mathsf{vl} \to \mathit{Values}_v$, $\mathcal{L}_v : \mathsf{tm} \to \Lambda_v$

specification. I provide the details in figure 3.4. ifbr and apply are constants which are not used in the translation $\mathcal{H}_v$, but arise during evaluation of translated Λ_v-terms in FORUM.

There are some rather subtle issues in the translation in figure 3.4. For instance, on what basis do I choose abs $\lambda x.\, x$ over $((\lambda u.\, u\ )(\mathsf{abs}\ \lambda x.\, x))$ as the encoding of $\lambda x.\, x$? The choice comes from the fact that there are unique $\beta\eta$-long normal forms in FORUM terms. So I pick the $\beta\eta$-long normal form as *the* encoding of the given Λ_v term. The translations from FORUM to Λ_v, and vice versa, in figures 3.5 and 3.4 respectively, are straightforward recursions on the structure of the terms.

I would like to evaluate with the translated programs. Hence, I need to check whether substitution commutes with the two translations $\mathcal{H}_v$ and $\mathcal{L}_v$. The following lemma states these identities precisely; proofs are deferred to the appendix.

Lemma 3.1 *Let $M \in \Lambda_v$, $V \in Values_v$, N and U be FORUM terms of types* tm *and* vl *respectively :*

- $\mathcal{H}_v(M[x := V]) \; = \; \mathcal{H}_v(M)[x := \phi_v(V)]$.
- $\mathcal{L}_v(N[x := U]) \; = \; \mathcal{L}_v(N)[x := \psi_v(U)]$.

Armed with the precise definitions, I am now in a position to define the evaluator and the *correspondence theorem.* Keeping in mind that Λ_v is to be evaluated left-to-right, and in fact, considering this as a part of the specification of Λ_v operational semantics, I would like to specify the evaluator such that left-to-right evaluation is enforced on implementations complying with my specifications. Hence, a natural choice is to use a continuation-passing-style semantics for Λ_v [Plo76, Rey72]. How is my specification then different from the standard continuation-passing-style semantics for Λ_v [Plo76, Rey72]? The main point is that just the translation of programs to continuation-passing semantics does not yield the operational semantics. One also needs a strategy for executing the resulting programs. As my specifications are logic programs, search for cut-free proofs corresponds to computation — I get an evaluator for Λ_v from very simple clauses.

The evaluator is presented as a set of universally quantified clauses defining the meaning of the non-logical constants I used in translating Λ_v into FORUM. (eval M K) is a two place predicate, the first argument being of type tm and the second being of type $\mathsf{vl} \rightarrow o$, the type of continuations, with the intended meaning that the term M is to be evaluated with K being the continuation. The computational paradigm is that I evaluate M, and whatever is its result, I pass it to K which then completes the evaluation. The evaluator $\mathcal{E}_v$ is defined in figure 3.6. The order of evaluation for the terms in Λ_v does not matter as far as one is concerned only with the values produced by the evaluations. In this sense, specifying the exact evaluation order may seem to be an overkill. However, the failure

$$
\begin{array}{rcll}
(\mathsf{eval}\ \langle V\rangle\ K) & \circ\!- & (K\,V) & \\
(\mathsf{eval}\ (\mathsf{app}\ M\ N)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{eval}\ N\ \lambda u.\,(\mathsf{apply}\ v\ u\ K))) & \\
(\mathsf{apply}\ (\mathsf{abs}\ R)\ U\ K) & \circ\!- & (\mathsf{eval}\ (R\,U)\ K) & \\
(\mathsf{eval}\ (f\ M\ N)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{eval}\ N\ \lambda u.\,(K\ \underline{f\ v\ u}))) & f \in \mathcal{O} \\
(\mathsf{eval}\ (\mathsf{cond}\ M\ N\ P)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{eval}\ (\mathsf{ifbr}\ v\ N\ P)\ K)) & \\
(\mathsf{eval}\ (\mathsf{ifbr}\ \mathsf{true}\ M\ N)\ K) & \circ\!- & (\mathsf{eval}\ M\ K) & \\
(\mathsf{eval}\ (\mathsf{ifbr}\ \mathsf{false}\ M\ N)\ K) & \circ\!- & (\mathsf{eval}\ N\ K) & \\
(\mathsf{eval}\ (\mathsf{letval}\ R\ M)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{eval}\ (R\ v)\ K)) & \\
(\mathsf{eval}\ (\mathsf{letfun}\ R_1\ R_2)\ K) & \circ\!- & (\mathsf{eval}\ (R_1\ (\mathsf{abs}\ \lambda x.\,\mathsf{letfun}\ (\lambda f.\,R_2\ f\ x)\ R_2))\ K) &
\end{array}
$$

Figure 3.6: Clauses in $\mathcal{E}_v$ - the evaluator for Λ_v

to produce a value may be for two reasons. Firstly, the evaluator may get stuck, or the arithmetic operators may be undefined for some values, and secondly, the evaluation may never terminate. Suppose P is a divergent program. The program $(+\ P\ (\div\ 5\ 0))$ will have an infinite evaluation tree under left-to-right evaluation, whereas under right-to-left evaluation it will result in a finite failure caused by a division by $\mathsf{0}$ error.

Now I have to define when a term evaluates to a value in FORUM. As computation corresponds to search for cut-free proofs, the definition will involve statements about existence of proofs of sequents in FORUM.

Definition 3.2 [Evaluating Λ_v terms in FORUM] $M : \mathsf{tm}$ evaluates to $V : \mathsf{vl}$, written as $\mathsf{eval}_v(M, V)$, if

$$\Sigma_v : \mathcal{E}_v\,;\ \longrightarrow\ \forall K : \mathsf{vl} \to o.\,(K\,V) \multimap (\mathsf{eval}\ M\ K)$$

is provable in FORUM. ■

I compare the evaluation of programs using $\mathcal{E}_v$ to the natural semantics evaluator to highlight some key aspects of $\mathcal{E}_v$. Lets look at the computation of $(\mathsf{app}\ P\ Q)$ and $(M\ N)$ where P and Q are $\mathcal{H}_v(M)$ and $\mathcal{H}_v(N)$ respectively.

In figure 3.7 I use the *backchain* rule. This is essentially a composite rule in which I choose a clause from the evaluator clauses such that a right-hand side formula unifies with the head of

$$\frac{\Sigma_v, K : \mathcal{E}_v\,;\,(K\,V) \longrightarrow (\mathsf{eval}\ P\ \lambda v.\,(\mathsf{eval}\ Q\ \lambda u.\,(\mathsf{apply}\ v\ u\ K)))}{\Sigma_v, K : \mathcal{E}_v\,;\,(K\,V) \longrightarrow (\mathsf{eval}\,(\mathsf{app}\ P\ Q)\ K)}$$

$$\frac{M \Downarrow \lambda x.\,L \quad N \Downarrow U \quad L[x := U] \Downarrow V}{M\ N \Downarrow V}$$

Figure 3.7: One step in the evaluation of $(M\ N)$

the clause. The justification of the rule follows with use of $\multimap$ -L followed by the observation that the right sub-proof of $\multimap$ -L will trivially follow from *initial*, as the head of the clause and right-hand side formulas unified. The key point of the above example is the role of the continuation. The natural semantic proof has three sub-proofs, whereas the $\mathcal{E}_v$ proof is linear — this corresponds to the idea that the evaluation order is completely specified. Furthermore, notice the structure of the continuation $\lambda v.\,(\mathsf{eval}\ Q\ \lambda u.\,(\mathsf{apply}\ v\ u\ K))$ — this term encodes the fact that the value of P, $(\mathsf{abs}\ R)$, will be bound to v, and then Q will be evaluated and its value, W, will be bound to u, and finally $(R\ W)$ will be evaluated. Hence, continuations provide notation within the syntax for the incomplete parts of natural semantics evaluation trees. This capability of representing incomplete proofs within the syntax plays a crucial role in the specification of exceptions and callcc. In fact, it seems that it is problematic to specify callcc in the natural semantics framework because of this deficiency in its syntax.

$\mathcal{E}_v$ and the natural semantics in figure 3.2 are two specifications of Λ_v. I prove that the two specifications are identical to the extent that the values computed are identical. The complete proof is deferred to the appendix.

Theorem 3.3 (Correspondence theorem for Λ_v) *For all closed Λ_v terms M and values V,*

$M \Downarrow V$ *if and only if* $\mathsf{eval}_v(\mathcal{H}_v(M), \phi_v(V))$.

The proof of theorem 3.3 is rather interesting. Going from left-to-right I induct on the evaluation tree for the term, *i.e.*, I build computations for larger terms using computations of the subterms. I illustrate the general strategy by showing the case for $M = (N\ P)$.

Suppose $(N\ P) \Downarrow V$, then the only way this may happen is by the use of rule for application terms, which implies that

- $N \Downarrow \lambda x.\, Q$,
- $P \Downarrow U$ and
- $Q[x := U] \Downarrow V$.

The evaluation trees of N, P and $Q[x := U]$ are smaller than the evaluation tree of $(N\ P)$. Let $N_1 =_{def} \mathcal{H}_v(N)$, $P_1 =_{def} \mathcal{H}_v(P)$, $Q_1 =_{def} \mathcal{H}_v(Q)$, $U_1 =_{def} \phi_v(U)$ and $V_1 =_{def} \phi_v(V)$. By induction hypothesis I get proofs δ_1, δ_2 and δ_3 in FORUM respectively for

- $\Sigma_v, K_1 : \mathcal{E}_v\, ;\, (K_1\, (\mathsf{abs}\ \lambda x.\, Q_1)) \longrightarrow (\mathsf{eval}\ N_1\ K_1)$,
- $\Sigma_v, K_2 : \mathcal{E}_v\, ;\, (K_2\, U_1) \longrightarrow (\mathsf{eval}\ P_1\ K_2)$ and
- $\Sigma_v, K_3 : \mathcal{E}_v\, ;\, (K_3\, V_1) \longrightarrow (\mathsf{eval}\ Q_1[x := U_1]\ K_3)$.

In the proof δ_3 I use lemma 3.1 to rewrite $\mathcal{H}_v(Q[x := U])$ as $Q_1[x := U_1]$. Using the above proofs I need to construct a proof for the sequent

$$\Sigma_v, K : \mathcal{E}_v\, ;\, (K\, V_1) \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ N_1\ P_1)\ K).$$

I show below how to construct the required proof. To keep the proof readable, I do not write $\mathcal{E}_v$ in the intuitionistic context and the signature which is Σ_v, K in all the sequents shown in the proof. In the proof let $C_1 =_{def} \lambda v.\, (\mathsf{eval}\ P_1\ \lambda u.\, (\mathsf{apply}\ v\ u\ K))$ and $C_2 =_{def} \lambda u. (\mathsf{apply}\ (\mathsf{abs}\ \lambda x. Q_1)\ u\ K)$. It is interesting to note that the $CutL$ rules are needed exactly at those points in the computation when a term passes its values to its continuation. The $CutL$ passes the value of N_1 to C_1, the continuation of N_1.

$$\dfrac{\dfrac{(K\, V) \overset{\gamma}{\longrightarrow} (\mathsf{eval}\ P_1\ C_2) \quad (C_1\, (\mathsf{abs}\ \lambda x.\, Q_1)) \overset{\sigma_1}{\longrightarrow} (\mathsf{eval}\ N_1\ C_1)}{(K\, V) \longrightarrow (\mathsf{eval}\ N_1\ C_1)}\ CutL}{(K\, V) \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ N_1\ P_1)\ K)}\ backchain$$

γ is constructed as shown below. The *CutL* passes the value of P_1 to C_2, the continuation of P_1.

$$\dfrac{\dfrac{(K\,V) \xrightarrow{\sigma_3} (\mathsf{eval}\; Q_1[x := U_1]\; K)}{(K\,V) \longrightarrow (\mathsf{apply}\; (\mathsf{abs}\; \lambda x.\, Q_1)\; U_1\; K)}\; \mathit{backchain} \qquad (C_2\, U_1) \xrightarrow{\sigma_2} (\mathsf{eval}\; P_1\; C_2)}{(K\,V) \longrightarrow (\mathsf{eval}\; P_1\; C_2)}\; \mathit{CutL}$$

Each σ_i, $i \in [1, 3]$ is constructed from δ_i using *CutS*. Note that

$$\Sigma_v, K_1, K : \mathcal{E}_v\, ;\, (K_1\, (\mathsf{abs}\; \lambda x.\, Q_1)) \longrightarrow (\mathsf{eval}\; N_1\; K_1)$$

is provable by δ_1 as $\Sigma_v, K_1 \subset \Sigma_v, K_1, K$. Hence σ_1 can be completed by cutting on K_1 with C_1. Similarly, one can build σ_2 and σ_3.

This completes the proof for the case $M = (N\ P)$. The important point is that proofs in FORUM are built using various *cut rules* rather than analyzing the structure of the proof, as one would expect for the left-to-right direction. Computation in FORUM is represented by search for *cut-free proofs*. However, since the proofs constructed above have cuts, I am implicitly using the cut-elimination theorem for FORUM. In the other direction, I analyze the proofs in FORUM and construct the natural semantics evaluation trees from FORUM proofs. The proofs are detailed in chapter A.

3.2 Λ_{vs} — State in UML

In this section I specify Λ_{vs}, Λ_v extended with state. Λ_v is a higher-order functional language where values are associated to variables via λ binding. For example, let val $x = 2$ in M associates the value 2 with the variable x in the term M. The salient property of such variable bindings is that it *cannot* be changed - x will remain bound to 2 throughout the evaluation of M. Although in principle, one can program only with λ bindings, in practice there are many situations where one would like to *update* the binding of a variable. For example, in Λ_v, I cannot write a function, inc, such that it takes a dummy argument and

$$
\begin{array}{rll}
M ::= & \ldots & (\Lambda_{vs}) \\
& \mid \textsf{ref}\ M & \\
& \mid \textsf{deref}\ M & \\
& \mid \textsf{asg}\ M\ M & \\
\\
E ::= & \ldots & (\mathit{EvCont}_{vs}) \\
& \mid E[\textsf{ref}\ [\,]] & \\
& \mid E[\textsf{deref}\ [\,]] & \\
& \mid E[\textsf{asg}\ [\,]\ M] & \\
& \mid E[\textsf{asg}\ V\ [\,]] &
\end{array}
$$

Figure 3.8: Syntax for Λ_{vs}

generates a new number every time it is called. Assignable variables introduce a notion of *state* in the programming language. The state of the computation is the current binding for the assignable variables. As I can update the binding for the assignable variables, the state may change during the computation.

The syntax for Λ_{vs} is formally defined in figure 3.8. The definitions of M and E in figure 3.1 are extended with the clauses in figure 3.8 to obtain Λ_{vs} and EvCont_{vs}, respectively. The definition for V in figure 3.1 remains unchanged for Values_{vs}. However, the non-terminal M ranges over the extended definition. The result of evaluating $(\textsf{ref}\ M)$ is a fresh location in the state which is bound to the value of M. $(\textsf{deref}\ M)$ evaluates M to a location and then returns the contents of the state at that location. $(\textsf{asg}\ M\ N)$ assigns the value of N to the location resulting from evaluating M, if that location is already defined in the current state. The $\textsf{inc}$ function described above may be implemented in Λ_{vs} as:

$$
\begin{array}{l}
\textsf{let val}\ x = \textsf{ref}\ 0\ \textsf{in} \\
\qquad \lambda y.\, (\lambda z.\, \textsf{deref}\ x)\ (\textsf{asg}\ x\ (+\ (\textsf{deref}\ x)\ 1))
\end{array}
$$

Evaluating M in store S may create a new location, say l. What should I do with l? One approach would be to extend S with l, and let l be in the state at the end of evaluating M. Another approach would be to treat l as a variable local to M, which must not be in the state at the end of evaluating M. While the first approach results in a global view

$$\frac{}{\langle l, S_0\rangle \Downarrow \langle l, S_0\rangle}\; l \in \mathsf{dom}(S_0)$$

$$\frac{\langle M, S_0\rangle \Downarrow \langle V, S_1\rangle}{\langle \mathsf{ref}\ M, S_0\rangle \Downarrow \langle l, S_1[l \mapsto V]\rangle}\; l \notin \mathsf{dom}(S_1)$$

$$\frac{\langle M, S_0\rangle \Downarrow \langle l, S_1\rangle}{\langle \mathsf{deref}\ M, S_0\rangle \Downarrow \langle S_1(l), S_1\rangle}\; l \in \mathsf{dom}(S_1)$$

$$\frac{\langle M, S_0\rangle \Downarrow \langle l, S_1\rangle \quad \langle N, S_1\rangle \Downarrow \langle V, S_2\rangle}{\langle \mathsf{asg}\ M\ N, S_0\rangle \Downarrow \langle \bullet, S_2[l \mapsto V]\rangle}\; l \in \mathsf{dom}(S_1)$$

Figure 3.9: Natural Semantics specification for the new constructs in Λ_{vs}

of state as taken by SML [HMT89], the second approach is adopted by block-structured languages like ALGOL [Rey81a, Rey81b]. State is thought of as a finite function from *Vars* to $Values_{vs}$. $\mathsf{dom}(S)$ for a state S is the set of variables which are bound in the state, *i.e.* the set of assignable variables currently defined. If $l \in \mathsf{dom}(S)$ then $S[l \mapsto V]$ denotes the state which maps l to V. If $l \notin \mathsf{dom}(S)$ then $S[l \mapsto V]$ denotes a new state, say S_1, such that $\mathsf{dom}(S_1)$ is the union of $\mathsf{dom}(S)$ and $\{l\}$, and $S_1(l) = V$.

I specify Λ_{vs} in natural semantics in the style of [HMT89]. The specification for the new constructs is in figure 3.9. In the evaluation of ($\mathsf{ref}\ M$) the side condition — $l \notin S_1$ — causes the creation of a new location in the state which is bound to V. deref reads the value of the state at a location l; the side condition $l \in S_1$ ensures that S_1 is defined for l. ($\mathsf{asg}\ M\ N$) redefines the binding of S_2 at the location l to be V, only if l is a variable already defined in S_2. Note that all currently defined assignable variables are treated as values. The changes to state are cumulative, *e.g.*, in the rule for ($\mathsf{asg}\ M\ N$), evaluation of M results in the state S_1 which is then passed along as the starting state for the computation of N. If I define some new locations in the process of evaluating M, they will be in $\mathsf{dom}(S_1)$, and hence finally in $S_2[l \mapsto V]$. Thus the rules specify a *global view of state.*

Unfortunately, the evaluator for Λ_{vs} is not simply the union of rules in figure 3.9 and figure 3.2. Although no new rules are needed for the constructs of Λ_v due to the addition of state, the existing rules in figure 3.2 need to be modified. *In this sense, the specification of Λ_v is not modularly extended to the specification of Λ_{vs}.* However, the damage is mild

$$\begin{aligned}
\mathsf{cell} &: \mathsf{tm} \to \mathsf{tm} \\
\mathsf{read} &: \mathsf{tm} \to \mathsf{tm} \\
\mathsf{write} &: \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \\
\mathsf{get} &: \mathsf{vl} \to (\mathsf{vl} \to o) \to o \\
\mathsf{set} &: \mathsf{vl} \to \mathsf{vl} \to (\mathsf{vl} \to o) \to o
\end{aligned}$$

Figure 3.10: Constants for translating Λ_{vs} terms

compared to the situation in section 3.3, where extending Λ_v with exceptions will create new rules for the constructs in Λ_v. The modification to the rules in figure 3.2 is obtained by applying the *state convention* in section 3.3 to states. For example

$$\frac{M \Downarrow \lambda x.\, P \quad N \Downarrow U \quad P[x := U] \Downarrow V}{M\ N \Downarrow V}$$

is considered to be an abbreviated form of

$$\frac{\langle M, S_0\rangle \Downarrow \langle \lambda x.\, P, S_1\rangle \quad \langle N, S_1\rangle \Downarrow \langle U, S_2\rangle \quad \langle P[x := U], S_2\rangle \Downarrow \langle V, S_3\rangle}{\langle M\ N, S_0\rangle \Downarrow \langle V, S_3\rangle}$$

Thus the natural semantics specification for Λ_{vs} is obtained by taking the rules in figure 3.9 and the rules obtained by applying the state convention to the rules in figure 3.2. I will freely use the abbreviated form of the natural semantics rules, because using the state convention I can always recover the full form.

I want to specify state in FORUM not as a finite function, but as some form of concurrent computation. I think of every location in the state as a separate process storing a value, which interacts with its environment only via read and write messages. Interaction on a read message causes the process to transmit the value it stores to the environment. Interaction on a write message causes the process to accept a value from the environment which replaces the value it stores. Adopting the paradigm used in section 2.1 to specify $HO\pi$-calculus in FORUM, I directly specify the above process-style reading of state in FORUM.

The signature for the translation, Σ_{vs}, is the union of Σ_v and the constants in figure 3.10.

$$
\begin{array}{rcl}
(\mathsf{eval}\ (\mathsf{cell}\ M)\ K) & \multimap & (\mathsf{eval}\ M\ \lambda v.\,\forall P,l.\,\mathsf{getC}(P,l) \Rightarrow \mathsf{setC}(P,l) \Rightarrow [(K\,l)\,\wp\,(P\,v)]) \\
(\mathsf{eval}\ (\mathsf{read}\ M)\ K) & \multimap & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{get}\ v\ K)) \\
(\mathsf{eval}\ (\mathsf{write}\ M\ N)\ K) & \multimap & (\mathsf{eval}\ M\ \lambda v.\,(\mathsf{eval}\ N\ \lambda u.\,(\mathsf{set}\ v\ u\ K)))
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & \mathsf{getC}(P,l) =_{def} \forall K,U.\,[(\mathsf{get}\ l\ K)\,\wp\,(P\,U)] \multimap [(K\,U)\,\wp\,(P\,U)] \\
 & \mathsf{setC}(P,l) =_{def} \forall K,V,U.\,[(\mathsf{set}\ l\ V\ K)\,\wp\,(P\,U)] \multimap [(K\,\bullet)\,\wp\,(P\,V)]
\end{array}
$$

Figure 3.11: Specification in FORUM for new constructs in Λ_{vs}

I define translations $\phi_{vs} : \mathit{Values}_{vs} \rightarrow \mathsf{vl}$, $\mathcal{H}_{vs} : \Lambda_{vs} \rightarrow \mathsf{tm}$, $\psi_{vs} : \mathsf{vl} \rightarrow \mathit{Values}_{vs}$ and $\mathcal{L}_{vs} : \mathsf{tm} \rightarrow \Lambda_{vs}$ in the appendix. For example, $\mathcal{H}_{vs}(\mathsf{ref}\ M) = \mathsf{cell}\ \mathcal{H}_{vs}(M)$. The following lemmas regarding the translations and substitution are proved in the appendix.

Lemma 3.4 *Let $M \in \Lambda_{vs}$, $V \in \mathit{Values}_{vs}$, N and U be a FORUM terms of types* tm *and* vl *respectively:*

- $\mathcal{H}_{vs}(M[x := V]) = \mathcal{H}_{vs}(M)[x := \phi_{vs}(V)]$.
- $\mathcal{L}_{vs}(N[x := U]) = \mathcal{L}_{vs}(N)[x := \psi_{vs}(U)]$.

The evaluator for Λ_{vs}, $\mathcal{E}_{vs}$, is the union $\mathcal{E}_v$ and the clauses in figure 3.11. $(\mathsf{cell}\ M)$ is evaluated by first evaluating M to a value V. Next, a new predicate P is created and placed in the environment storing V as $(P\,V)$. The process identifier, l, for the process P, is passed to the continuation K. Along with the creation of the process, two more clauses are introduced, namely getC and setC. getC specifies the handling of read messages passed along the process identifier l, while setC specifies the handling of write messages passed along the identifier l. A read message, $(\mathsf{get}\ l\ K)$, synchronizes with the process identified by l, reads its value, and passes the value to the continuation K. A write message, $(\mathsf{set}\ l\ V\ K)$, synchronizes with the process identified by l, sets its value to V, and passes the token $\bullet$ to the continuation K, indicating the successful completion of the write message. $(\mathsf{read}\ l)$ issues the read message to the process identified by l, and $(\mathsf{write}\ l\ V)$ issues the write message to the process identified by l. I now prove the *correspondence theorem* between the two specifications, namely $\mathcal{E}_{vs}$ and natural semantics, for Λ_{vs}. Before I can do this, I have to define when a term evaluates to a value in FORUM and translate state into FORUM.

Remark 3.5 *I define notation that I use in describing evaluation in FORUM. Let S be a state, and m be the number of elements in* $\mathsf{dom}(S)$.

- $P_S =_{def} P_1, \ldots, P_m$.
- $l_S =_{def} l_1, \ldots, l_m,\ l_i \in \mathsf{dom}(S), i \in [1, m]$.
- $\Sigma_S =_{def} \{P_1, \ldots, P_m\} \cup \{l_1, \ldots, l_m\}$.
- $\mathsf{CL}_S =_{def} \{\mathsf{getC}(P_1, l_1), \ldots, \mathsf{getC}(P_m, l_m)\} \cup \{\mathsf{setC}(P_1, l_1), \ldots, \mathsf{setC}(P_m, l_m)\}$.
- $\Gamma_S =_{def} (P_1\, V_1) \wp \ldots \wp (P_m\, V_m)$.
- $V_i =_{def} \phi_{vs}(S(l_i)),\ i \in [1, m]$.
- $FV(S(l_i)) \subset \mathsf{dom}(S),\ i \in [1, m]$.
- *A term M is closed in a state S, written as* $\mathsf{close}(S, M)$, *if* $FV(S(l_i)) \subset \mathsf{dom}(S)$, $i \in [1, m]$ *and* $FV(M) \subset \mathsf{dom}(S)$.

Definition 3.6 [Translating state into FORUM] The translation of state S, written as S^o, is a FORUM term of type $o \rightarrow o$.

$$S^o =_{def} \lambda u : o. \forall\, P_S, l_S.\ \mathsf{getC}(P_1, l_1) \Rightarrow \mathsf{setC}(P_1, l_1) \Rightarrow \ldots \Rightarrow \mathsf{getC}(P_m, l_m) \Rightarrow \mathsf{setC}(P_m, l_m) \Rightarrow [u \wp \Gamma_S]$$

■

The application of a term M to S^o would require that the free variables in M be named apart from the bound variables in S^o including l_S. I abuse notation because I want the location names to be "captured" by the substituion. Suppose S is $\lambda u. \forall l.\ \mathsf{getC}(P, l) \Rightarrow$

$\mathsf{setC}(P, l) \Rightarrow [u \wp \Gamma_S]$ and M is $\langle l \rangle$, then $(S\,\langle l \rangle)$ is $\forall l.\, \mathsf{getC}(P, l) \Rightarrow \mathsf{setC}(P, l) \Rightarrow [\langle l \rangle \wp \Gamma_S]$. Note that l is free in $\langle l \rangle$ but in $(S\,\langle l \rangle)$ it gets captured by the universal quantification on l. This abuse of syntax comes in very handy, and should not be confusing.

Definition 3.7 [Evaluating Λ_{vs} terms in FORUM] Let M be a Λ_{vs}-term, and S_0 be a state such that $\mathsf{close}(S_0, M)$. M with S_0 evaluates to V with S_1, written $\mathsf{eval}_{vs}(M, S_0, V, S_1)$, if

$$\Sigma_{vs} : \mathcal{E}_{vs}\,;\ \longrightarrow \forall K : \mathsf{vl} \rightarrow o.\ {S_1}^o(K\,V) \multimap\ {S_0}^o(\mathsf{eval}\ \mathcal{H}_{vs}(M)\ K)$$

is provable in FORUM. ■

Theorem 3.8 (Correspondence theorem for Λ_{vs}) *Let M be a Λ_{vs} term, and S_0 a state such that* $\mathsf{close}(S, M)$.

$\langle M, S_0 \rangle \Downarrow \langle V, S_1 \rangle$ *if and only if* $\mathsf{eval}_{vs}(M, S_0, V, S_1)$

The proof of theorem 3.8 is deferred to the appendix. It is along the same lines as the proof for theorem 3.3. Given the fact that Λ_{vs} has an imperative state, it is not a priori clear whether I can use Cut rules to compose proofs - the richer proof-theory of FORUM permits me to use Cut rules essentially because environments are maintained using logical constants. I illustrate the proof strategy going left-to-right when $\langle \mathsf{asg}\ M\ N, S_0 \rangle \Downarrow \langle \bullet, S_2[l \mapsto V] \rangle$. The last rule in the evaluation tree has to be for asg, which implies that

- $\langle M, S_0 \rangle \Downarrow \langle l, S_1 \rangle$,
- $\langle N, S_1 \rangle \Downarrow \langle V, S_2 \rangle$, and
- $l \in \mathsf{dom}(S_2)$.

The evaluation trees of M and N are smaller than the evaluation tree of $(\mathsf{asg}\ M\ N)$. Let $M_1 =_{def} \mathcal{H}_{vs}(M)$, $N_1 =_{def} \mathcal{H}_{vs}(N)$, $S_3 =_{def} S_2[l \mapsto V]$ and $V_1 =_{def} \phi_{vs}(V)$. By induction hypothesis I get proofs δ_1 and δ_2 in FORUM respectively for

- $\mathsf{eval}_{vs}(M, S_0, l, S_1)$, and
- $\mathsf{eval}_{vs}(N, S_1, V, S_2)$.

Further, note that $l \in \Sigma_{S_2}$. Using the above proofs, I need to construct a proof for the sequent

$$\Sigma_v, K, \Sigma_{S_3} : \mathcal{E}_{vs}\,;\, S_3{}^o(K\,\bullet) \longrightarrow S_0{}^o(\mathsf{eval}\ (\mathsf{write}\ M_1\ N_1)\ K).$$

Below, I construct the required proof. To keep the proof readable, I do not write $\mathcal{E}_{vs}$ in the intuitionistic context, and Σ_{vs}, K in the signature part of the sequent, as these parts are present in all the sequents in the part of the proof shown. Furthermore, I do not show the introductions of Σ_S and CL_S in the sequents below, because these can be deduced from the context. Let $C_1 =_{def} \lambda v.\,(\mathsf{eval}\ N_1\ \lambda u.\,(\mathsf{set}\ v\ u\ K))$ and $C_2 =_{def} \lambda u.\,(\mathsf{set}\ l\ u\ K)$. In the proof, I first do a bunch of right $\forall$ introductions followed by a bunch of right $\Rightarrow$ introductions to introduce Σ_{S_0} and CL_{S_0}. Next, I backchain on the clause for asg clause, and then I do a *CutL*. I am left with the construction of γ and σ_1.

$$\dfrac{\dfrac{\dfrac{\overset{\gamma}{S_3{}^o(K\,\bullet) \longrightarrow S_1{}^o(\mathsf{eval}\ N_1\ C_2)} \quad \overset{\sigma_1}{S_1{}^o(C_1\,l) \longrightarrow (\mathsf{eval}\ M_1\ C_1) \wp \Gamma_{S_0}}}{S_3{}^o(K\,\bullet) \longrightarrow (\mathsf{eval}\ M_1\ C_1) \wp \Gamma_{S_0}}}{S_3{}^o(K\,\bullet) \longrightarrow (\mathsf{eval}\ (\mathsf{write}\ M_1\ N_1)\ K) \wp \Gamma_{S_0}}\ \mathit{backchain}}{\dfrac{\vdots, \forall R, \Rightarrow R}{S_3{}^o(K\,\bullet) \longrightarrow S_0{}^o(\mathsf{eval}\ (\mathsf{write}\ M_1\ N_1)\ K)}}$$

The construction of γ is given below. I first do a bunch of right $\forall$ introductions followed by a bunch of right $\Rightarrow$ introductions to introduce Σ_{S_1} and CL_{S_1}. Next, I do a *CutL*.

$$\dfrac{\dfrac{\overset{\gamma_1}{S_3{}^o(K\,\bullet) \longrightarrow S_2{}^o(\mathsf{set}\ l\ V_1\ K)} \quad \overset{\sigma_2}{S_2{}^o(C_2\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ C_2) \wp \Gamma_{S_1}}}{S_3{}^o(K\,\bullet) \longrightarrow (\mathsf{eval}\ N_1\ C_2) \wp \Gamma_{S_1}}}{\dfrac{\vdots, \forall R, \Rightarrow R}{S_3{}^o(K\,\bullet) \longrightarrow S_1{}^o(\mathsf{eval}\ N_1\ C_2)}}$$

Suppose $\mathsf{dom}(S_2) =_{def} l_1, \ldots, l_n$. As $l \in \mathsf{dom}(S_2)$, for some $i \in [1, n]$, l is l_i. Construction of γ_1 is shown below. I start the proof with a bunch of right $\forall$ introductions followed by a bunch of right $\Rightarrow$ introductions to introduce Σ_{S_2} and CL_{S_2}. Now I backchain over the $\mathsf{setC}(P_i, l_i)$ to change the value stored as P_i to be V_1. I am left with the construction of γ_2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\gamma_2}{S_3{}^o(K\,\bullet) \;\longrightarrow\; (K\,\bullet) \wp (P_i\, V_1) \wp (P_1\, U_1) \ldots \wp (P_n\, U_n)}}{S_3{}^o(K\,\bullet) \;\longrightarrow\; (\mathsf{set}\ l_i\ V_1\ K) \wp (P_i\, U_i) \wp (P_1\, U_1) \ldots \wp (P_n\, U_n)}}{S_3{}^o(K\,\bullet) \;\longrightarrow\; (\mathsf{set}\ l_i\ V_1\ K) \wp (P_1\, U_1) \wp \ldots (P_i\, U_i) \ldots \wp (P_n\, U_n)}}{\dfrac{\vdots, \forall R, \Rightarrow R}{S_3{}^o(K\,\bullet) \;\longrightarrow\; S_2{}^o(\mathsf{set}\ l_i\ V_1\ K)}}$$

The construction of γ_2 is rather interesting. I start a proof with a bunch of left $\forall$ introductions. The purpose of these is to identify the location names in the final state with the locations created during the evaluation of the term. Next, I perform a bunch of left $\Rightarrow$ introductions. The purpose of these is to constrain the substitution for the P variables. Essentially, the getC and setC clauses for each location are matched off against each other. I then reorganize the memory on the right-hand side to match the order in Γ_{S_3}, and use identity.

$$\dfrac{\dfrac{\dfrac{}{(K\,\bullet) \wp \Gamma_{S_3} \;\longrightarrow\; (K\,\bullet) \wp \Gamma_{S_3}}}{\vdots}}{\dfrac{(K\,\bullet) \wp \Gamma_{S_3} \;\longrightarrow\; (K\,\bullet), \wp (P_i\, V_1) \wp (P_1\, U_1) \ldots \wp (P_n\, U_n)}{\dfrac{\vdots, \forall L, \Rightarrow L}{S_3{}^o(K\,\bullet) \;\longrightarrow\; (K\,\bullet) \wp \Gamma_{S_3}}}}$$

The construction of σ_1 and σ_2 from δ_1 and δ_2 is straightforward. Note that *CutL* is only used when terms pass values to their continuations. The novelty of this proof lies in the way the cell l is updated to store V_1. The proof makes crucial use of multiple heads to synchronize between the set instruction and P_i, the process identified by l. Values can be updated by backchaining, because memory cells are linear objects in FORUM. When a cell synchronizes with a message from the environment, it is *consumed*, and thus needs to be refreshed. When refreshed, it may be updated as the setC clause does.

3.3 Λ_{ve} — Exceptions in UML

In this subsection I specify Λ_{ve} - Λ_v with exceptions. Exceptions are a very important feature of any programming language, and especially indispensable in programs which accept inputs from users or external programs. The openin function of SML is a good example. Given the name of a file, the function opens a stream for reading the data in the file. However, if there does not exist a file with the specified name, then openin is faced with an *exceptional situation*. At this point openin has two acceptable strategies. The first strategy is that openin returns some value indicating the fact that the file does not exist - the path taken by C. The shortcoming of this strategy is that the programmer must check the value returned by openin to see whether the file was actually opened or not. If the programmer forgets to do so, one may get some obscure error in a possibly unrelated part of the program, and then have to trace the error back to the non-existence of the file. The second strategy is to send a signal to the function which invoked openin. Now, if the programmer does not check for the signal intentionally, it will cause the program to stop, and print an error message saying that the specified file did not exist. However, if not checking the signal was an oversight, then the error will be reported as being caused by the fact the specified file did not exist, and hence is easily detectable.

The SML exception mechanism makes possible the second of the two choices outlined above. There is no restriction on the number of exceptions that one may have in a SML program — exceptions can be created on the fly. Furthermore, exception handlers are scoped, *i.e.* I can declare a handler for an exception for any given sub-part of my program. These features make the exception mechanism of SML rich and elaborate. I begin by extending the syntax of Λ_v for exceptions in figure 3.12. The definitions of M, V and E in figure 3.1 are extended with the clauses in figure 3.12 to obtain Λ_{ve}, Values_{ve} and EvCont_{ve} respectively. Declared exception names are also values. For this, I introduce a new countable syntactic class of *ExnNames* ranged over by l. The result of a computation now may not be a value, *e.g.* an uncaught exception. I introduce a new syntax class called answers for this purpose, and the natural semantic clause $M \Downarrow A$ will now read as M evaluates to the answer A. $[\mathsf{pk}\ l\ V]$

$$
\begin{array}{lll}
M ::= & \ldots & (\Lambda_{ve}) \\
 & \mid \textsf{exception}\ l\ M & l \in ExnNames \\
 & \mid \textsf{handle}\ M\ M\ M & \\
 & \mid \textsf{raise}\ M\ M & \\
V ::= & \ldots & (Values_{ve}) \\
 & \mid l & l \in ExnNames \\
 & & \\
A ::= & V & (Answers_{ve}) \\
 & \mid [\textsf{pk}\ l\ V] & l \in ExnNames \\
 & & \\
E ::= & \ldots & (EvCont_{ve}) \\
 & \mid E[\textsf{raise}\ [\,]\ M] & \\
 & \mid E[\textsf{raise}\ V\ [\,]] & \\
 & \mid E[\textsf{exception}\ x\ [\,]] & \\
 & \mid E[\textsf{handle}\ M\ [\,]\ M] & \\
 & \mid E[\textsf{handle}\ M\ V\ [\,]] & \\
 & \mid E[\textsf{handle}\ [\,]\ V\ V] &
\end{array}
$$

Figure 3.12: Syntax for Λ_{ve}

is a called a packet, raising the exception l with the value V.

($\textsf{exception}\ l\ M$) binds the exception name l, in the scope of M. It is entirely conceivable to write a program such as ($\textsf{exception}\ l$ (($\textsf{exception}\ l\ N$) M)). This scoping of exceptions creates the need for renaming of exception names, *i.e.* α-conversion. This problem is handled in [HMT89] by evaluating the exception l to a new exception name, and carrying this binding around in the environment. ($\textsf{raise}\ l\ V$) indicates that the exceptional circumstance as indicated by l has occurred, and the function handling this exception should be called with the value V. ($\textsf{handle}\ M\ l\ N$) declares that during the evaluation of M, if the exception l is raised and uncaught within M, then N will be the handling function. Furthermore, if the evaluation of M to a value is completed without raising l, then N is removed as the function handling the exception l. This semantics of installing handlers locally provides flexibility during programming.

I illustrate the point with the following example. Let $\textsf{xn}$ be some exception name resulting from the declaration of some exception, and P some function defined in the environment.

$$
\begin{array}{l}
(\textsf{handle}(\textsf{if}\ (=\ x\ 0) \\
\qquad\qquad (\textsf{handle}\ (P\ x)\ \textsf{xn}\ N_1) \\
\qquad\qquad M \\
\qquad)\textsf{xn}\ N_2)
\end{array}
$$

Now, which handler is used for the exception exn depends upon the value of x. Furthermore, if P is called and in P handlers are installed for xn, then those handlers take precedence over N_1. There are some elements of dynamic binding in the mechanism for determining which handler catches a raised exception. It is rather tricky to specify this exception mechanism because of the above mentioned considerations.

The specification of exceptions in [HMT89] is presented in a very slick manner. The specification proceeds in two stages. First, the rules needed for the new constructs are specified as in figure 3.13. The spirit of the rules remains the same as the rule for Λ_v. The evaluation of a term is a result of the synthesis of the evaluations of its subterms. However, now I need to keep track of the exception names which have been declared thus far, which means that I need to carry along a state in the evaluation rules. The operational semantics as presented here differs from [HMT89] to the extent that I use substitution instead of maintaining closures.

Clearly, the mere addition of the above rules to $\mathcal{E}_v$ is not enough to specify Λ_{ve}. One problem is that the propagation of exception names has to be handled for all the existing rules for Λ_v terms. For this purpose, the *state convention* is adopted in [HMT89], which tells us how to restore exception states in a rule which omits them. According to this convention, if a rule is presented as

$$\frac{M_1 \Downarrow V_1 \quad \ldots \quad M_n \Downarrow V_n}{M \Downarrow V}$$

then its full form is intended to be

$$\frac{}{\langle l, Ex\rangle \Downarrow \langle l, Ex\rangle}\ l \in ExnNames$$

$$\frac{\langle M[x := l], Ex_0 \cup l\rangle \Downarrow \langle V, Ex_1\rangle}{\langle \textsf{exception}\ x\ M, Ex_0\rangle \Downarrow \langle V, Ex_1\rangle}\ l \notin Ex_0$$

$$\frac{\langle M, Ex_0\rangle \Downarrow \langle l, Ex_1\rangle \quad \langle N, Ex_1\rangle \Downarrow \langle V, Ex_2\rangle}{\langle \textsf{raise}\ M\ N, Ex_0\rangle \Downarrow \langle [\textsf{pk}\ l\ V], Ex_2\rangle}$$

$$\frac{\langle N, Ex_0\rangle \Downarrow \langle l, Ex_1\rangle \quad \langle P, Ex_1\rangle \Downarrow \langle W, Ex_2\rangle \quad \langle M, Ex_2\rangle \Downarrow \langle V, Ex_3\rangle}{\langle \textsf{handle}\ M\ N\ P, Ex_0\rangle \Downarrow \langle V, Ex_3\rangle}\ *$$

$$\frac{\begin{array}{c}\langle N, Ex_0\rangle \Downarrow \langle l, Ex_1\rangle \\ \langle P, Ex_1\rangle \Downarrow \langle W, Ex_2\rangle \\ \langle M, Ex_2\rangle \Downarrow \langle [\textsf{pk}\ l\ U], Ex_3\rangle \\ \langle (W\ U), Ex_3\rangle \Downarrow \langle V, Ex_4\rangle\end{array}}{\langle \textsf{handle}\ M\ N\ P, Ex_0\rangle \Downarrow \langle V, Ex_4\rangle}$$

$$\frac{\langle N, Ex_0\rangle \Downarrow \langle l_0, Ex_1\rangle \quad \langle P, Ex_1\rangle \Downarrow \langle W, Ex_2\rangle \quad \langle M, Ex_2\rangle \Downarrow \langle [\textsf{pk}\ l_1\ U], Ex_3\rangle}{\langle \textsf{handle}\ M\ N\ P, Ex_0\rangle \Downarrow \langle [\textsf{pk}\ l_1\ U], Ex_3\rangle}\ l_0 \neq l_1,\ *$$

Figure 3.13: Natural Semantics specification for the new constructs in Λ_{ve}.

$$\frac{\langle M_1, Ex_0\rangle \Downarrow \langle V_1, Ex_1\rangle \quad \ldots \quad \langle M_n, Ex_{n-1}\rangle \Downarrow \langle V_n, Ex_n\rangle}{\langle M, Ex_0\rangle \Downarrow \langle V, Ex_n\rangle}$$

As I can always derive the full-form of a rule using the *state convention*, I will freely use the abbreviated versions of natural semantic rules from now on. The specification is still not complete. For example, what do I do if the evaluation of N in (raise M N) raises an exception? In fact, this question comes up in each and every rule specified in figure 3.2. Unfortunately, this leads to adding more clauses for all the rules. The description of the additional rules needed can be done concisely along the lines of [HMT89]. An *exception convention* defines new natural semantic rules based on the ones in figures 3.2 and 3.13, except for the rules labeled * in figure 3.13. Suppose the form of a rule is:

$$\frac{M_1 \Downarrow V_1 \quad \ldots \quad M_n \Downarrow V_n}{M \Downarrow V}$$

Then for every k, $1 \leq k \leq n$, such that V_k is not a packet, we add another rule of the form

:

$$\frac{M_1 \Downarrow V_1 \quad \ldots \quad M_k \Downarrow [\mathsf{pk}\ l\ V]}{M \Downarrow [\mathsf{pk}\ l\ V]}$$

For example, the rule for application will now result in the following three new rules.

$$\frac{M \Downarrow [\mathsf{pk}\ l\ V]}{M\ N \Downarrow [\mathsf{pk}\ l\ V]}$$

$$\frac{M \Downarrow \lambda x.\,P \quad N \Downarrow [\mathsf{pk}\ l\ V]}{M\ N \Downarrow [\mathsf{pk}\ l\ V]}$$

$$\frac{M \Downarrow \lambda x.\,P \quad N \Downarrow U \quad P[x := U] \Downarrow [\mathsf{pk}\ l\ V]}{M\ N \Downarrow [\mathsf{pk}\ l\ V]}$$

The natural semantics evaluator for Λ_{ve} is specified by the rules in figure 3.13, 3.2 and the ones created as a result of adopting the exception convention explained above. *Thus, the exception convention causes the number of rules in the evaluator to increase from fourteen to thirty-two*!

Let me present another way of looking at the operational semantics for exceptions. Suppose I am evaluating the term $E[\mathsf{handle}\ M\ l\ V]$ where ($\mathsf{handle}\ M\ l\ V$) is the redex that I am reducing currently, and E is the current evaluation context or the part of the program that will take the value of ($\mathsf{handle}\ M\ l\ V$) and complete the evaluation of the term, *i.e. the current continuation*. Further, let us suppose l and l_1 are two exception names, and no handler is installed for l_1. The evaluation of M, if it terminates, will yield

1. a value U_1, or

2. a packet $[\mathsf{pk}\ l\ U_2]$, or

3. a packet $[\mathsf{pk}\ l_1\ U_3]$.

In case 1, the computation continues with $E[U_1]$. In case 2, the computation continues with $E[V\,U_2]$. But what should happen in case 3? A signal, l_1, has been raised for which there is no handler. The only reasonable thing to do is to throw away the current continuation, *i.e.* E, and report to the top level or the next outer handler, that the exception l_1 was raised with the value U_3. *What is clear from the explanation is that exceptions can cause the computation to discard its current continuation up to a handler.*

Consider the evaluation of the term $E[\mathsf{handle}\ ((\mathsf{raise}\ l\ U)\ N)\ l\ V]$. The program will evaluate $(\mathsf{handle}\ ((\mathsf{raise}\ l\ U)\ N)\ l\ V)$ causing the handler V to be installed for the exception l, and then proceed with the evaluation of $((\mathsf{raise}\ l\ U)\ N)$, which in turn will cause it to evaluate $(\mathsf{raise}\ l\ U)$. Evaluating $(\mathsf{raise}\ l\ U)$ will result in a packet $[\mathsf{pk}\ l\ U]$, which can only be handled by V. However, notice that the answer computed by $(\mathsf{raise}\ l\ U)$ is passed not to the current continuation at that point, but rather to the continuation at the time when the handler V was installed! *Thus exceptions can also change the current continuation.*

My point is that a natural operational reading of the evaluation mechanism involves the idea of continuations, a concept which has no direct representation in the syntax of natural semantics. The clever presentation in [HMT89] is a way of overcoming this shortcoming of natural semantics. However, one has to pay a price for extending natural semantics to cope with exceptions — *the blow up in the number of rules for Λ_v terms from eight to twenty.* Of particular concern is the fact that the evaluator has to be redefined for the term constructs of Λ_v, *e.g.* one has four rules for application now. In this sense, the specification of exceptions in [HMT89] is *not* modular. An exception mechanism very similar to that of UML has been specified in [WF91] using term rewriting machines. The specification is indeed modular — specifications for the existing term constructs do not change when the language with exceptions is considered. However, the style in [WF91] introduces two new notions of contexts, one used to maintain the scope of exceptions and the other used to match a raised signal with its handler — the basic intuition about manipulation of continuations is not brought out very clearly.

The specification of Λ_{ve} in FORUM follows the same pattern as the specification of Λ_v. I

$$\begin{array}{rcl}
\mathsf{ex} & : & \mathsf{ext} \to \mathsf{vl} \\
\mathsf{exn} & : & (\mathsf{ext} \to \mathsf{tm}) \to \mathsf{tm} \\
\mathsf{install} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \\
\mathsf{signal} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \\
\mathsf{uncaught} & : & \mathsf{ext} \to \mathsf{vl} \to o
\end{array}$$

Figure 3.14: Constants for translating Λ_{ve} terms

$$\begin{array}{rcll}
\phi_{ve}(l) & = & (\mathsf{ex}\ l) & l \in \mathit{ExnNames} \\
\\
\mathcal{H}_{ve}(\mathsf{exception}\ l\ M) & = & \mathsf{exn}\ \lambda l.\, \mathcal{H}_{ve}(M) & \\
\mathcal{H}_{ve}(\mathsf{handle}\ M\ N\ P) & = & \mathsf{install}\ \mathcal{H}_{ve}(M)\, \mathcal{H}_{ve}(N)\, \mathcal{H}_{ve}(P) & \\
\mathcal{H}_{ve}(\mathsf{raise}\ M\ N) & = & \mathsf{signal}\ \mathcal{H}_{ve}(M)\, \mathcal{H}_{ve}(N) &
\end{array}$$

Figure 3.15: Translating the new construct of Λ_{ve} to FORUM

introduce a new type ext, the type of exception names in FORUM. First, I translate Λ_{ve} terms into FORUM syntax using Σ_v augmented with the constants in figure 3.14. For example, $\mathsf{install} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm}$ is the target of the translation of handle terms. $[\mathsf{pk}\ l\ V]$ is translated using $\mathsf{uncaught} : \mathsf{ext} \to \mathsf{vl} \to o$. I define translations $\phi_{ve} : \mathit{Values}_{ve} \to \mathsf{vl}$, $\mathcal{H}_{ve} : \Lambda_{ve} \to \mathsf{tm}$, $\psi_{ve} : \mathsf{vl} \to \mathit{Values}_{ve}$ and $\mathcal{L}_{ve} : \mathsf{tm} \to \Lambda_{ve}$. I show the translation from Λ_{ve} to FORUM for the new constructs in figure 3.15. The complete translations are deferred to the appendix. The following lemmas regarding the translations and substitutions are proved in the appendix.

Lemma 3.9 *Let $M \in \Lambda_{ve}$, $V \in \mathit{Values}_{ve}$, N and U be a FORUM terms of types* tm *and* vl *respectively:*

- $\mathcal{H}_{ve}(M[x := V]) = \mathcal{H}_{ve}(M)[x := \phi_{ve}(V)]$.
- $\mathcal{L}_{ve}(N[x := U]) = \mathcal{L}_{ve}(N)[x := \psi_{ve}(U)]$.

Instead of specifying the semantics of catching raised exceptions in the style of [HMT89], I take the route of using the continuations explicitly. Consequently, I have to manage explicitly the installation and removal of handlers, and the matching of raised exceptions with handlers. I find this description to be more enlightening because it highlights the manipulation of continuations by exceptions, and explains the maintenance of exception handlers separately. I represent the exceptions via a predicate, exnst, which takes a list of exception handlers as an argument. An exception handler is a term, (pkt $l\ V\ K$), where l is the exception name, V is the function installed as the handler, and K is the continuation which will be invoked if this handler is chosen. To maintain exnst I need predicates push and pop to add and remove handlers from the list of exception handlers. Further, I need a predicate lookup to search the list of handlers.

(lookup $l\ V$) searches the list X in (exnst X) for the first packet whose exception name is l. The search program for lookup would therefore need to check whether two exception names are equal or not, *i.e.* I need an inequality predicate for exception names. Specifying inequality in the presence of $\forall$ is a rather delicate matter. Suppose I am able to prove $\forall x, y.\ x \neq y$, how can I then use it? Obviously, instantiating the proof for $x = c$ and $y = c$, where c is a constant, will lead to inconsistencies. [HSH90, SH93, Gir92] analyze such situations. However, the results are not very conclusive as yet.

The solution I adopt is to take the type ext to be nat of section 2.1, and as shown in section 2.1, inequality between numbers can be specified in FORUM. I generate new exception names using the paradigm of inc in section 2.1. The signature for the various constants is given in figure 3.16. In the translation in figure 3.15, l is translated to (ex l). Given the new interpretation of ext, I have to redefine the translation. As *ExnNames* is a countable set, there is an isomorphism between *ExnNames* and ext. Hence, by abuse of notation I let l denote a term of type ext, which denotes the number to which l is mapped by the isomorphism.

The signature for the additional constants is given in figure 3.16. The signature for specifying Λ_{ve}, Σ_{ve}, is defined as the set consisting of the elements of Σ_v, the constants in figure 3.16, and the constants in figure 3.14.

$$
\begin{aligned}
\mathsf{pkt} &: \mathsf{ext} \to \mathsf{vl} \to (\mathsf{vl} \to o) \to \mathsf{packet} \\
\mathsf{push} &: \mathsf{ext} \to \mathsf{vl} \to (\mathsf{vl} \to o) \to \mathsf{tm} \to o \\
\mathsf{pop} &: o \to o \\
\mathsf{nil} &: \mathsf{packet\ list} \\
:: &: \mathsf{packet} \to \mathsf{packet\ list} \to \mathsf{packet\ list} \\
\mathsf{exnst} &: \mathsf{packet\ list} \to o \\
\mathsf{isexn} &: \mathsf{vl} \to (\mathsf{ext} \to o) \to o \\
\mathsf{z} &: \mathsf{ext} \\
\mathsf{s} &: \mathsf{ext} \to \mathsf{ext} \\
\mathsf{neq} &: \mathsf{ext} \to \mathsf{ext} \to o \\
\mathsf{sigctr} &: \mathsf{ext} \to o
\end{aligned}
$$

Figure 3.16: Constants for exception management in FORUM

I consider handle as an instruction to store an exception handler with the current continuation, and raise as an instruction to search for the most recently installed handler for the raised exception. (handle M l V) will cause V to be *pushed* on the stack of handlers as the handler for l before the execution of M begins, and if the execution of M terminates without raising l to the handler V, then V is removed from the stack of exception handlers. (raise l V) causes the system to look in the exception stack from top to bottom for a handler for l. If a handler is found for l, then control is passed to the handler and its continuation, or else the exception returns to the top-level as an uncaught exception. The specification for the new constructs along with the specification for the auxiliary non-logical constants is presented in figure 3.17. The evaluator, $\mathcal{E}_{ve}$, is defined to be the universal closure of the clauses in figure 3.17, the clauses in $\mathcal{E}_v$.

The clauses in figure 3.17 highlight various aspects of the exception mechanism. The fact that new exceptions can be created on the fly is implicit in the usage of the sigctr to generate a new constants for exception names. The way continuations are handled by exceptions is made explicit by handle storing the current continuation along with the handler in the exception stack. The search for a matching exception explains how exceptions can cause a program to discard its current continuation and reinstate the continuation stored with the handler. The issue of locally installing and removing handlers is a separate concern, managed by the stack like maintenance of exnst via push and pop. The clause for lookup

$$
\begin{array}{rcl}
(\mathsf{eval}\ (\mathsf{exn}\ R)\ K) \wp (\mathsf{sigctr}\ Y) & \multimap & (\mathsf{eval}\ (R\,Y)\ K) \wp (\mathsf{sigctr}\ (\mathsf{s}\ Y)) \\
(\mathsf{eval}\ (\mathsf{install}\ M\ N\ P)\ K) & \multimap & (\mathsf{eval}\ N\ \lambda v : \mathsf{vl}.\ \mathsf{isexn}\ v\ \lambda w : \mathsf{ext}. \\
 & & \quad (\mathsf{eval}\ P\ \lambda u : \mathsf{vl}.\ (\mathsf{push}\ w\ u\ K\ M))) \\
(\mathsf{eval}\ (\mathsf{signal}\ M\ N)\ K) & \multimap & (\mathsf{eval}\ M\ \lambda v : \mathsf{vl}.\ \mathsf{isexn}\ v\ \lambda w : \mathsf{ext}. \\
 & & \quad (\mathsf{eval}\ N\ \lambda u : \mathsf{vl}.\ (\mathsf{lookup}\ v\ u))) \\
 & & \\
(\mathsf{isexn}\ (\mathsf{ex}\ L)\ K) & \multimap & (K\ L) \\
(\mathsf{push}\ L\ V\ K\ M) \wp (\mathsf{exnst}\ X) & \multimap & (\mathsf{eval}\ M\ \lambda v.\ (\mathsf{pop}\ (K\ v))) \\
 & & \wp (\mathsf{exnst}\ (\mathsf{pkt}\ L\ V\ K) :: X) \\
(\mathsf{pop}\ P) \wp (\mathsf{exnst}\ (\mathsf{pkt}\ L\ V\ K) :: X) & \multimap & P \wp (\mathsf{exnst}\ X) \\
(\mathsf{neq}\ (\mathsf{s}\ X)\ \mathsf{z}) & \multimap & \top \\
(\mathsf{neq}\ \mathsf{z}\ (\mathsf{s}\ X)) & \multimap & \top \\
(\mathsf{neq}\ (\mathsf{s}\ X)\ (\mathsf{s}\ Y)) & \multimap & (\mathsf{neq}\ X\ Y) \\
(\mathsf{lookup}\ L\ V) \wp (\mathsf{exnst}\ (\mathsf{pkt}\ L\ U\ K) :: X) & \multimap & (\mathsf{apply}\ U\ V\ K) \wp (\mathsf{exnst}\ X) \\
(\mathsf{lookup}\ L\ V) \wp (\mathsf{exnst}\ (\mathsf{pkt}\ L_1\ U\ K) :: X) & \multimap & (\mathsf{neq}\ L\ L_1) \\
 & & \otimes ((\mathsf{lookup}\ L\ V) \wp (\mathsf{exnst}\ X)) \\
(\mathsf{lookup}\ L\ V) \wp (\mathsf{exnst}\ \mathsf{nil}) & \multimap & (\mathsf{uncaught}\ L\ (\mathsf{signal}\ L\ \langle V \rangle)) \wp (\mathsf{exnst}\ \mathsf{nil})
\end{array}
$$

Figure 3.17: Specification in FORUM for new constructs in Λ_{ve}

uses a new constant $\otimes$. Although the use of $\otimes$ makes the intention clear, it is not essential because in FORUM

$$:;\ A \multimap B \multimap C \longrightarrow (A \otimes B) \multimap C$$

is provable.

The key point is that neither do I introduce any new clauses for the constructs in Λ_v, nor do I modify the existing clauses in Λ_v. The specification of the exception mechanism is a modular extension to the specification for Λ_v. The fact that I can specify the stack in the environment using a logical connective, *i.e.* $\wp$, is rather important. If I attempt to specify Λ_{ve} using non-logical constants in the spirit of abstract machines, then I will not have to add new clauses for Λ_v, but I will have to redefine the clauses for Λ_v to explicitly carry along the exception stack. *However, because of the proof rules for $\wp$, I am able to keep the stack passively in the environment, and interact with it only when I have to deal with exceptions.* The rich proof theory of FORUM provides me the tools to specify the maintenance of the environment in a *logical fashion.* I now state the *correspondence theorem* between the two specifications, namely $\mathcal{E}_{ve}$ and [HMT89]. If Ex is a set of exception names, let l_{Ex} be one plus the largest number to which any element of Ex is mapped. Further, $\mathcal{G}(l, P) =_{def} P \wp (\mathsf{exnst\ nil}) \wp (\mathsf{sigctr}\ l)$.

Definition 3.10 [Translating $Answers_{ve}$ to FORUM] $\mathcal{A}_{ve}$ translates terms in $Answers_{ve}$ to terms in FORUM of type o. Let K be a constant of type $\mathsf{vl} \rightarrow o$ in FORUM.

$$\begin{array}{rcl} \mathcal{A}_{ve}(V, K) & = & (K\, \phi_{ve}(V)) \\ \mathcal{A}_{ve}([\mathsf{pk}\ l\ V], K) & = & (\mathsf{uncaught}\ l\ \phi_{ve}(V)) \end{array}$$

■

Definition 3.11 [Evaluating Λ_{ve} terms in FORUM] Let M be a Λ_{ve}-term, $A \in Answers_{ve}$, and Ex_0 be an exception state such that all the free variables of M are in Ex_0. M in exception state Ex_0 evaluates to A with the exception state Ex_1, written as $\mathsf{eval}_{vs}(\mathcal{H}_{ve}(M), l_{Ex_0}, \mathcal{A}_{ve}(A, K), l_{Ex_1})$, if

$$\Sigma_{ve} : \mathcal{E}_{ve}\, ;\ \longrightarrow \forall K : \mathsf{vl} \rightarrow o.\ \mathcal{G}(l_{Ex_1}, \mathcal{A}_{ve}(A, K)) \multimap \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\ \mathcal{H}_{ve}(M)\ K))$$

is provable in FORUM. ■

Theorem 3.12 (Correspondence theorem for Λ_{ve}) *Let M be a Λ_{ve} term, Ex_0 be the exception state consisting of all the exception names in M, $A \in Answers_{ve}$, and Ex_1 be the exception state consisting of all the exception names in A. All the free variables of M are in Ex_0.*

$\langle M, Ex_0 \rangle \Downarrow \langle A, Ex_1 \rangle$ *if and only if* $\mathsf{eval}_{vs}(\mathcal{H}_{ve}(M), l_{Ex_0}, \mathcal{A}_{ve}(A, K), l_{Ex_1})$

The proof of theorem 3.12 is deferred to the appendix. The proof is along the same lines as the proof for theorem 3.3. The logical maintainence of environments permits me to use cuts to compose proofs. I illustrate the proof strategy going left to right when $\langle N\ P, E_0 \rangle \Downarrow \langle [\mathsf{pk}\ l\ V], E_3 \rangle$. One way this may happen is that

- $\langle N, E_0\rangle \Downarrow \langle \lambda x.\, Q, E_1\rangle$,
- $\langle P, E_1\rangle \Downarrow \langle U, E_2\rangle$ and
- $\langle Q[x := U], E_2\rangle \Downarrow \langle [\mathsf{pk}\ l\ V], E_3\rangle$.

The evaluation trees of N, P and $Q[x := U]$ are smaller than the evaluation tree of $(N\ P)$. Let $N_1 =_{def} \mathcal{H}_{ve}(N)$, $P_1 =_{def} \mathcal{H}_{ve}(P)$, $Q_1 =_{def} \mathcal{H}_{ve}(Q)$, $U_1 =_{def} \phi_{ve}(U)$, $V_1 =_{def} \phi_{ve}(V)$ and $A =_{def} (\mathsf{uncaught}\ l\ V)$. By induction hypothesis I get proofs δ_1, δ_2 and δ_3 in FORUM respectively for

- $\Sigma_v, K_1 : \mathcal{E}_v\, ;\, \mathcal{G}(l_{Ex_1}, (K_1\, (\mathsf{abs}\ \lambda x.\, Q_1))) \longrightarrow \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\ N_1\ K_1))$,
- $\Sigma_v, K_2 : \mathcal{E}_v\, ;\, \mathcal{G}(l_{Ex_2}, (K_2\, U_1)) \longrightarrow \mathcal{G}(l_{Ex_1}, (\mathsf{eval}\ P_1\ K_2))$, and
- $\Sigma_v, K_3 : \mathcal{E}_v\, ;\, \mathcal{G}(l_{Ex_3}, A) \longrightarrow \mathcal{G}(l_{Ex_2}, (\mathsf{eval}\ Q_1[x := U_1]\ K_3))$.

In the proof δ_3 I use lemma 3.9 to rewrite $\mathcal{H}_v(Q[x := U])$ as $Q_1[x := U_1]$. Using the above proofs I need to construct a proof for the sequent

$$\Sigma_v, K : \mathcal{E}_v\, ;\, \mathcal{G}(l_{Ex_3}, A) \longrightarrow \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\, (\mathsf{app}\ N_1\ P_1)\ K)).$$

I construct the required proof below. To keep the proof readable, I do not write $\mathcal{E}_{ve}$ in the intuitionistic context, and the signature which is Σ_{ve}, K. In the proof let $C_1 =_{def} \lambda v.\, (\mathsf{eval}\ P_1\ \lambda u.\, (\mathsf{apply}\ v\ u\ K))$, and $C_2 =_{def} \lambda u.\, (\mathsf{apply}\ (\lambda x.\, Q_1)\ u\ K)$. Note that $(C_1\, (\mathsf{abs}\ \lambda x.\, Q_1))$ is β equivalent to $(\mathsf{eval}\ P_1\ C_2)$, and that $(C_2\, U_1)$ is β equivalent to $(\mathsf{apply}\ (\mathsf{abs}\ \lambda x.\, Q_1)\ U_1\ K)$. The last proof rule is *CutL* in the construction below.

$$\dfrac{\dfrac{\mathcal{G}(l_{Ex_3}, A) \overset{\gamma}{\longrightarrow} \mathcal{G}(l_{Ex_1}, (\mathsf{eval}\ P_1\ C_2)) \quad \mathcal{G}(l_{Ex_1}, (\mathsf{eval}\ P_1\ C_2)) \overset{\sigma_1}{\longrightarrow} \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\ N_1\ C_1))}{\mathcal{G}(l_{Ex_3}, A) \longrightarrow \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\ N_1\ C_1))}}{\mathcal{G}(l_{Ex_3}, A) \longrightarrow \mathcal{G}(l_{Ex_0}, (\mathsf{eval}\, (\mathsf{app}\ N_1\ P_1)\ K))}\ \mathit{backchain}$$

To complete the proof I have to construct γ and σ_1. σ_1 is constructed from δ_1, by using *CutS* on K_1 with C_1. I show the construction of γ. I use a *CutL* followed by a *backchain* on the left sub-proof of *CutL*.

$$
\dfrac{\dfrac{\mathcal{G}(l_{Ex_3}, A) \ \stackrel{\sigma_3}{\longrightarrow}\ \mathcal{G}(l_{Ex_2}, (\mathsf{eval}\ Q_1[x := U_1]\ K))}{\mathcal{G}(l_{Ex_3}, A) \ \longrightarrow\ \mathcal{G}(l_{Ex_2}, (\mathsf{eval}\ C_2\ U_1))} \qquad \mathcal{G}(l_{Ex_2}, C_2\, U_1) \ \stackrel{\sigma_2}{\longrightarrow}\ \mathcal{G}(l_{Ex_1}, (\mathsf{eval}\ P_1\ C_2))}{\mathcal{G}(l_{Ex_3}, A) \ \longrightarrow\ \mathcal{G}(l_{Ex_1}, (\mathsf{eval}\ P_1\ C_2))}
$$

The construction of σ_2 and σ_3 is along the lines of σ_1, using δ_2 and δ_3, respectively. The surprising fact is that the structure of the proof construction given above, and the construction of $\sigma_i, i \in [1, 3]$ is identical to the construction given in the example for theorem 3.3. *The fact that I can use Cut rules to compose proofs and cut-elimination to obtain computations in the presence of exceptions is a convincing argument for the claim that $\mathcal{E}_{ve}$ is a declarative specification of Λ_{ve}.*

3.4 Λ_{vc} — Continuations in UML

In this subsection I specify Λ_{vc} - Λ_v with continuations. A *continuation* is that part of a program which must be evaluated to obtain the final value of the computation. For example, consider the evaluation of $(+\ (+\ 3\ 2)\ 4)$. When I am evaluating $(+\ 3\ 2)$, the remaining part of the computation is $(+\ \bullet\ 4)$. The idea being that the value of $(+\ 3\ 2)$ will replace $\bullet$, and then computation will start once again, or, in other words, $\lambda x.(+\ x\ 4)$ is the continuation. Informally, one may think of a continuation as a function which takes the value of the term being evaluated currently as its argument, and then continues the computation. If I can represent the continuation of a program in the program itself, then I can change the control flow of the program during its evaluation. Thus, continuations provide a functional means of representing the control flow of the program.

Continuations have been used widely in the semantics, compilation and design of programming languages [Gor79, Sto77, SW74, AJ89, App92, Ste78, RC86, SF92]. Many programming languages provide language constructs which allow the programmer to take control of the continuation of the program during the computation and manipulate it, *e.g.* callcc in SML of New Jersey, and call-with-current-continuation in scheme. Programming with continuations introduces a rich programming style. [Fri88, DH89, Wan80] implemented coroutines and process schedulers using scheme-style call-with-current-continuation. [Rep91] implemented CML, a concurrent higher order functional language with concurrency primitives, using callcc in SML of New Jersey.

Continuations are not a part of the core SML [HMT89]. The reasons for not providing language support for manipulating continuations are not presented in [HMT89]. However, given the obvious utility of continuations as a programming paradigm, I consider first-class continuations as an important aspect of a programming language. Moreover, SML of New Jersey does include callcc, a construct that allows the programmer to capture the current continuation. We saw in section 3.1 that exceptions can be specified in natural semantics, and that continuations are manipulated when exceptions are raised. Hence, some level of continuation manipulation can be specified in the natural semantics framework. Thus encouraged, I attempted to specify callcc in the natural semantics framework outlined in [HMT89]. The operational semantics of callcc is informally specified as — first evaluate M to V, and then apply V to whatever the current continuation is. Hence, to specify callcc I need to formalize the notion of *current continuation* within the syntax of the language. *Without changing the essential nature of the natural semantics framework it seemed impossible to specify* callcc.

In natural semantics, the continuation at any point of the computation is not a part of the syntax. For example, let us look at the evaluation for $(M\ N)$ in figure 3.2.

$$\frac{M \Downarrow \lambda x.\, P \quad N \Downarrow U \quad P[x := U] \Downarrow V}{M\ N \Downarrow V}$$

The continuation of M is a program which will take $\lambda x.\, P$, compute N to U and then compute $P[x := U]$ to V. This information is encoded in the above rule by the three separate hypotheses of the rule, and demanding that the hypotheses be read from left to right. If I want to capture the current continuation in the program, then I must be able to represent the continuation in the evaluation rule — it is not clear how to do this in natural semantics. *Exceptions also manipulate continuations — how can exceptions be specified in natural semantics?* Exceptions can be specified in natural semantics because of two

$$
\begin{array}{rll}
M ::= & \ldots & (\Lambda_{vc}) \\
& \mid \mathsf{callcc}\ M & \\
& \mid \mathsf{throw}\ M\ M & \\
\\
E ::= & \ldots & (EvCont_{vc}) \\
& \mid E[\mathsf{callcc}\ [\,]] & \\
& \mid E[\mathsf{throw}\ [\,]\ M] & \\
& \mid E[\mathsf{throw}\ V\ [\,]] &
\end{array}
$$

Figure 3.18: Syntax for Λ_{vc}

key reasons. Firstly, exceptions do not allow the programmer to capture the continuation as a part of the program. Secondly, exceptions do not allow the programmer to use a continuation more than once. Although the path taken in [CF94] is a beginning towards using a concurrent meta-theory with side-effects, it is however restricted by the fact that there are essentially two threads, the program evaluator and the administrator.

The syntax for Λ_{vc} is presented in figure 3.18. The definitions of M and E in figure 3.1 are extended with the clauses in figure 3.18 to obtain Λ_{vc} and $EvCont_{vc}$ respectively. The definition for V in figure 3.1 remains unchanged for $Values_{vc}$. However, the non-terminal M ranges over the extended definition. I think of ($\mathsf{callcc}\ V$) as creating a process out of the current continuation, called the *continuation process* — the process identifier for the created process is passed to V. Unlike processes created in concurrent languages, the continuation process does not start computing on its own, in fact it lies dormant. The process identified by l is invoked by ($\mathsf{throw}\ l\ U$). As Λ_{vc} is a sequential language, only one process should be active at any given time. Hence, the continuation process is dormant when created, and when a continuation process is invoked using throw, the evaluation thread terminates and the continuation process becomes the evaluation thread. I find this view of callcc to be very useful, especially in light of the fact that one of the most significant uses of callcc has been to implement CML in SML of New Jersey [Rep91]. Adopting the paradigm used in section 2.1 to specify the $HO\pi$-calculus in FORUM, I directly specify the above process-style reading of callcc in FORUM.

$$\begin{array}{rcl} \mathsf{catch} & : & \mathsf{tm} \rightarrow \mathsf{tm} \\ \mathsf{jump} & : & \mathsf{tm} \rightarrow \mathsf{tm} \rightarrow \mathsf{tm} \\ \mathsf{resume} & : & \mathsf{vl} \rightarrow \mathsf{vl} \rightarrow o \\ \mathsf{cont} & : & \mathsf{vl} \rightarrow (\mathsf{vl} \rightarrow o) \rightarrow o \end{array}$$

Figure 3.19: Constants for translating Λ_{vc} terms

$$\begin{array}{rcl} (\mathsf{eval}\ (\mathsf{catch}\ M)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\, \forall l.\, \mathsf{contC}(l, K) \Rightarrow [(\mathsf{apply}\ v\ l\ K) \wp (\mathsf{cont}\ l\ K)]) \\ (\mathsf{eval}\ (\mathsf{jump}\ M\ N)\ K) & \circ\!- & (\mathsf{eval}\ M\ \lambda v.\, (\mathsf{eval}\ N\ \lambda u.\, (\mathsf{resume}\ v\ u))) \end{array}$$

where $\quad \mathsf{contC}(l, K) =_{def} \forall U.\, [(\mathsf{resume}\ l\ U) \wp (\mathsf{cont}\ l\ K)] \circ\!- [(K\ U) \wp (\mathsf{cont}\ l\ K)]$

Figure 3.20: Specification in FORUM for new constructs in Λ_{vc}

The signature for the specification, Σ_{vc}, is defined to be Σ_v extended with the constants in figure 3.19, *e.g.* callcc is translated to catch. I define translations $\phi_{vc} : \mathit{Values}_{vc} \rightarrow \mathsf{vl}$, $\mathcal{H}_{vc} : \Lambda_{vc} \rightarrow \mathsf{tm}$, $\psi_{vc} : \mathsf{vl} \rightarrow \mathit{Values}_{vc}$ and $\mathcal{L}_{vc} : \mathsf{tm} \rightarrow \Lambda_{vc}$ in the appendix. For example, $\mathcal{H}_{vc}(\mathsf{callcc}\ M) = \mathsf{catch}\ \mathcal{H}_{vc}(M)$. The following lemmas regarding the translations and substitutions are proved in the appendix.

Lemma 3.13 *Let $M \in \Lambda_{vc}$, $V \in \mathit{Values}_{vc}$, N and U be a FORUM terms of types* tm *and* vl *respectively:*

- $\mathcal{H}_{vc}(M[x := V]) = \mathcal{H}_{vc}(M)[x := \phi_{vc}(V)]$.
- $\mathcal{L}_{vc}(N[x := U]) = \mathcal{L}_{vc}(N)[x := \psi_{vc}(U)]$.

The evaluator $\mathcal{E}_{vc}$ for Λ_{vc} is the universal closure of the clauses in figure 3.20 along with the clauses in $\mathcal{E}_v$. Below, I outline the evaluation of $(\mathsf{eval}\ (\mathsf{catch}\ M)\ K)$ to highlight the novel features of this specification.

- To evaluate $(\mathsf{eval}\ (\mathsf{catch}\ M)\ K)$, I first evaluate M to a value V, which is passed to the continuation $\lambda v.\, \forall l.\, \mathsf{contC}(l, K) \Rightarrow [(\mathsf{apply}\ v\ l\ K) \wp (\mathsf{cont}\ l\ K)]$.

- Evaluation of $\forall l.\, \mathsf{contC}(l, K) \Rightarrow [(\mathsf{apply}\ V\ l\ K) \wp (\mathsf{cont}\ l\ K)]$ creates *a new name* $l : \mathsf{vl}$ and *makes* $\mathsf{contC}(l, K)$ *a part of the evaluator.*

- Finally, I proceed with the evaluation of $(\mathsf{apply}\ V\ l\ K) \wp (\mathsf{cont}\ l\ K)$.

Placing $(\mathsf{cont}\ l\ K)$ in the environment of $(\mathsf{apply}\ V\ l\ K)$ is the creation of the continuation process as a dormant entity — l is the process identifier for the continuation process $(\mathsf{cont}\ l\ K)$. Every time I create a continuation process, I introduce the $\mathsf{contC}(l, K)$ clause which associates the process identifier l to $(\mathsf{cont}\ l\ K)$, the continuation processes. When $(\mathsf{resume}\ l\ U)$ synchronizes with $(\mathsf{cont}\ l\ K)$, it results in the evaluation of $(K\,V)$ with $(\mathsf{cont}\ l\ K)$ in the environment. $(\mathsf{jump}\ M\ N)$ terminates the current evaluation thread and invokes one of the dormant continuation processes. *The main novelty lies in the way I can create new entities in the environment,* e.g. $(\mathsf{cont}\ l\ K)$. *The created processes lie in the environment dormant, and can be activated by passing control explicitly to them,* e.g. *the* $\mathsf{contC}(l, K)$ *clause.* This specification provides a view of catch and jump as a restricted form of concurrency where it is possible to have more than one process. However, only one of the processes can compute at a given time. Given this basic understanding, it is natural that callcc in SML of New Jersey was the basis of the CML implementation in SML of New Jersey [Rep91] and call-with-current-continuation was used to implement co-routines and process schedulers [Fri88, DH89, Wan80].

I would like to define when a term M evaluates to a value V. When computation starts, there may be continuation processes defined in the environment in which M is to be evaluated. As the computation of M proceeds, new continuation processes may be created. Hence, the value V will have a *continuation state*, *i.e.* the continuation processes associated with it. In this sense continuations behave like a mutable store. *Hence, to evaluate Λ_{vc} terms one has to carry the continuation processes and the corresponding process identifiers created thus far, just as in evaluating programs with assignable variables one must carry along the mutable store.* A continuation state, written as Γc, possibly subscripted, is defined to be a finite function from variables to $EvCont_{vc}$. $\mathsf{dom}(\Gamma c)$ denotes the domain of the function Γc. Let $l_{\Gamma c}$ denote a list of the variables in $\mathsf{dom}(\Gamma c)$ in any order. *A key difference between*

$$
\begin{array}{rcl}
\mathcal{C}_{vc}([\,], K) & = & K \\
\mathcal{C}_{vc}(E[f\,[\,]\,M], K) & = & \lambda v.\,(\mathsf{eval}\ \mathcal{H}_{vc}(M)\ \lambda u.\ K'\ \underline{f\ v\ u}) \\
\mathcal{C}_{vc}(E[f\ V\ [\,]], K) & = & \lambda v.\ K'\ \underline{f\ \phi_{vc}(V)\ v} \\
\mathcal{C}_{vc}(E[[\,]\ M], K) & = & \lambda v.\,(\mathsf{eval}\ \mathcal{H}_{vc}(M)\ \lambda u.\,(\mathsf{apply}\ v\ u\ K')) \\
\mathcal{C}_{vc}(E[V\ [\,]], K) & = & \lambda v.\,(\mathsf{apply}\ \phi_{vc}(V)\ v\ K') \\
\mathcal{C}_{vc}(E[\mathsf{if}\ [\,]\ M\ N], K) & = & \lambda v.\,(\mathsf{eval}\ (\mathsf{ifbr}\ v\ \mathcal{H}_{vc}(M)\ \mathcal{H}_{vc}(N))\ K') \\
\mathcal{C}_{vc}(E[\mathsf{let\ val}\ x = [\,]\ \mathsf{in}\ M], K) & = & \lambda v.\,(\mathsf{eval}\ \mathcal{H}_{vc}(M)[x := v]\ K') \\
\mathcal{C}_{vc}(E[\mathsf{callcc}\ [\,]], K) & = & \lambda v.\,\forall l.\ \mathsf{contC}(l, K') \Rightarrow [(\mathsf{apply}\ v\ l\ K')\wp(\mathsf{cont}\ l\ K')] \\
\mathcal{C}_{vc}(E[\mathsf{throw}\ [\,]\ M], K) & = & \lambda v.\,(\mathsf{eval}\ \mathcal{H}_{vc}(M)\ \lambda u.\,(\mathsf{resume}\ v\ u)) \\
\mathcal{C}_{vc}(E[\mathsf{throw}\ V\ [\,]], K) & = & \lambda v.\,(\mathsf{resume}\ \phi_{vc}(V)\ v)
\end{array}
$$

where $$K' =_{def} \mathcal{C}_{vc}(E, K : \mathsf{vl} \rightarrow o)$$

Figure 3.21: $\mathcal{C}_{vc}$, translation of $EvCont_{vc}$ to FORUM terms of type $\mathsf{vl} \rightarrow o$

the continuation state and the mutable store is that the process identifiers in Γc *cannot be reassigned.* Consequently, Γc is cycle-free, *i.e.* there is an ordering on $l_{\Gamma c}$, say $l_{i_1} \ldots l_{i_m}$ where $\mathsf{dom}(\Gamma c)$ has m elements, such that for all $l \in FV(\Gamma c(l_{i_j}))$, $l < l_{i_j}$, $j \in [1, m]$. In particular, $l \notin FV(\Gamma c(l))$, for all $l \in \mathsf{dom}(\Gamma c)$ and $FV(\Gamma c(l_{i_1})) = \emptyset$. By a continuation state I will henceforth mean a cycle-free continuation state. $\mathcal{C}_{vc}$ in figure 3.21 translates $E \in EvCont_{vc}$ to the terms in FORUM of type $\mathsf{vl} \rightarrow o$.

Definition 3.14 [Translating continuation state into FORUM] The translation of Γc, written as Γc^o, is a FORUM term of type $o \rightarrow o$.

$$
\begin{aligned}
\Gamma c^o \ =_{def}\ \lambda u : o.\forall\, l_{\Gamma c}\ & \mathsf{contC}(l_1, K_1) \Rightarrow \ldots \Rightarrow \\
& \mathsf{contC}(l_m, K_m) \Rightarrow [u\,\wp\,(\mathsf{cont}\ L\ K_1)\,\wp\,\ldots\,\wp\,(\mathsf{cont}\ l_m\ K_m)]
\end{aligned}
$$

Where m is the number of elements of $\mathsf{dom}(\Gamma c)$ and $K_i =_{def} \mathcal{C}_{vc}(\Gamma c(l_i), K)$, $i \in [1, m]$. ■

Note the substitution of a term M for u in Γc^o is treated like the substitution a term in in the translation of a state. In particular, the free variables $l_{\Gamma c}$ in M will get "captured" by the substitution.

$$(\mathsf{eval}\ \ (\mathsf{catch}\ M)\ K)\ \wp\,(\mathsf{exnst}\ X)\ \circ\!-$$
$$\qquad (\mathsf{eval}\ M\ \lambda v.\forall P,l.\ \mathsf{contC}(l,K,X) \Rightarrow [(\mathsf{apply}\ v\ l\ K)\ \wp\,(\mathsf{cont}\ l\ K\ X)\ \wp\,(\mathsf{exnst}\ X)])$$

where, $\mathsf{contC}(l, K, X) =_{def}$
$$\forall U,Y.\ [(\mathsf{resume}\ l\ U)\ \wp\,(\mathsf{cont}\ l\ K\ X)\ \wp\,(\mathsf{exnst}\ Y)]\ \circ\!-\ [(K\ U)\ \wp\,(\mathsf{cont}\ l\ K\ X)\ \wp\,(\mathsf{exnst}\ X)]$$

Figure 3.22: Specification for callcc in the presence of exceptions

Definition 3.15 [Evaluating Λ_{vc} terms in FORUM] Let M be a Λ_{vc}-term, and Γc_1 be a continuation state such that all free variables of M are in $\mathsf{dom}(\Gamma c_1)$. M with the continuation state Γc_1 evaluates to V with the continuation state Γc_2, written $\mathsf{eval}_{vc}(M, \Gamma c_1, V, \Gamma c_2)$ if

$$\Sigma_{vc} : \mathcal{E}_{vc}\,;\ \longrightarrow\ \forall K : \mathsf{vl} \to o.\ \Gamma c_2{}^o(K\ V)\ \multimap\ \Gamma c_1{}^o(\mathsf{eval}\ \mathcal{H}_{vc}(M)\ K)$$

is provable in FORUM. ■

A further issue arises when callcc and throw need to be specified in the presence of exceptions, *i.e.* I am extending Λ_{ve} with first-class continuations. The question is regarding the status of exception handlers when I *throw to a continuation process.* In SML of New Jersey, the exception handlers are stored along with the continuation. Consequently, if I throw to a continuation, then along with restoring the saved continuation as the current continuation, I install the saved exception handlers as the currently installed exception handlers. Since the definition of the continuation is extended to include exception handlers, the clause for catch is changed to treat exceptions. *The achievement here is that only the specifications for* catch *and* contC *need to be changed; everything else remains the same.* The new specifications for catch and contC are given in figure 3.22. Note that the type of cont is $\mathsf{vl} \to (\mathsf{vl} \to o) \to \mathsf{packet\ list} \to o$. The integration of first-class continuations will be done in detail later, when I put together the specification for UML using all the pieces. As [HMT89] does not specify callcc and throw, I cannot prove equivalence between the two semantics for Λ_{vc}.

3.5 UML — Putting it together

In this section I provide the specification for UML — Λ_v with state, exceptions and continuations. The syntax for UML is obtained by putting together the syntax in figures 3.1, 3.8, 3.12, and 3.18. A natural semantic specification for Λ_v with state and exceptions is obtained in two steps. First, take the rules in figures 3.2, 3.9, and 3.13. Now, apply the state and exception convention to the rules thus obtained. The resulting set of rules is the natural semantic specification for Λ_v with state and exceptions. As discussed in section 3.4, I do not have a natural semantics specification of first-class continuations.

In FORUM, it is possible to specify UML as evidenced by the specifications of its parts in sections 3.1, 3.2, 3.3 and 3.4. The signature of the translation, Σ_{ml}, is the union of Σ_v, Σ_{vs}, Σ_{ve} and Σ_{vc}. The translations from UML to FORUM and back, $\phi : \mathit{Values} \rightarrow \mathsf{vl}$, $\mathcal{H} : \Lambda \rightarrow \mathsf{tm}$, $\mathcal{A} : \mathit{Answers} \rightarrow (\mathsf{vl} \rightarrow o) \rightarrow o$, $\mathcal{C} : \mathit{EvCont} \rightarrow (\mathsf{vl} \rightarrow o)$, $\psi : \mathsf{vl} \rightarrow \mathit{Values}$, and $\mathcal{L} : \mathsf{tm} \rightarrow \Lambda$, are obtained by putting the translations for the fragments together. The definitions are deferred to the appendix. The evaluator for UML, $\mathcal{E}$ is defined as the universal closure of the clauses in figures 3.6, 3.11, 3.17, and 3.20 — *I put together the different modules specifying different pieces of FORUM.* A *configuration* is a triplet of continuation state, state and the exception names with respect to which a UML term is evaluated. I now define the translation of a configuration in FORUM, and the evaluation for UML terms in FORUM.

Definition 3.16 [Translating configuration to FORUM] Let $C =_{def} \langle \Gamma c, S, Ex \rangle$ be a configuration, where Γc is a continuation state, S is a state and Ex a set of exception names. The translation of C, written as C^o, is a FORUM term of type $o \rightarrow o$.

$$
\begin{aligned}
&C^o =_{def} \\
&\lambda u : o. \forall\, P_S, l_S, p_{\Gamma c}.\ \mathsf{getC}(P_1, l_1) \Rightarrow \mathsf{setC}(P_1, l_1) \Rightarrow \ldots \Rightarrow \mathsf{getC}(P_n, l_n) \Rightarrow \mathsf{setC}(P_n, l_n) \Rightarrow \\
&\qquad\qquad\qquad\qquad \mathsf{contC}(p_1, K_1) \Rightarrow \ldots \Rightarrow \mathsf{contC}(p_m, K_m) \Rightarrow \\
&\qquad\qquad\qquad\qquad [u \wp \Gamma_S \wp (\mathsf{cont}\ p_1\ K_1) \wp \ldots \wp (\mathsf{cont}\ p_m\ K_m) \wp (\mathsf{sigctr}\ l_{Ex}) \wp (\mathsf{exnst}\ \mathsf{nil})]
\end{aligned}
$$

Where

- n is the number of elements in $\mathsf{dom}(\Gamma c)$, and m is the number of elements in $\mathsf{dom}(S)$.
- $K_i =_{def} \mathcal{C}(\Gamma c(p_i), K)$, $i \in [1, n]$.

■

The domain of a configuration $C =_{def} \langle \Gamma c, S, Ex \rangle$, written as $\mathsf{dom}(C)$, is the union of $\mathsf{dom}(\Gamma c)$, $\mathsf{dom}(S)$ and Ex. These are the variables which have been declared to be values by the configuration C. Note that analogous to the case of S^o and Γc^o, applying a term M to C^o causes the free occurances of l_S and $p_{\Gamma c}$ in M, if any, to get captured.

Definition 3.17 [Evaluating UML terms in FORUM] Let M be a UML term, $A \in Answers_{ve}$, and C_1 a configuration such that $FV(M) \subset \mathsf{dom}(C_1)$. M in the configuration C_1 evaluates to A in the configuration C_2, written as $\mathsf{eval}(M, C_1, A, C_2)$, if

$$\Sigma_{ml} : \mathcal{E}\,; \longrightarrow \forall K : \mathsf{vl} \rightarrow o.\, C^o \mathcal{A}(A, K) \multimap C^o(\mathsf{eval}\ M\ K)$$

is provable in FORUM. ■

The part of UML specified by natural semantics, Λ_{vse}, does not contain first-class continuations. If I restrict the above definitions to UML without callcc and throw, then I can prove a correspondence theorem between the two specifications. For this fragment I consider configurations in which the continuation state is empty. The proof is deferred to the appendix.

Theorem 3.18 (Correspondence theorem for Λ_{vse}) *Let $M \in \Lambda_{vse}$, $A \in Answers_{vse}$, $C_1 =_{def} \langle \emptyset, S_1, Ex_1 \rangle$ and $C_2 =_{def} \langle \emptyset, S_2, Ex_2 \rangle$. Further, M is closed with respect to C_1.*

$\langle M, C_1 \rangle \Downarrow \langle A, C_2 \rangle$ *if and only if* $\mathsf{eval}(\mathcal{H}(M), C_1, \mathcal{A}(A, K), C_2)$

Chapter 4

Program Equivalence for Λ_{vs} in FORUM

In this chapter, I study observational equivalence for Λ_{vs} programs in FORUM. A program is treated like a black-box — the only way to determine the behavior of a program is to give it some inputs and watch the output it generates. In this paradigm, two programs are equivalent if whenever they are given identical inputs they generate identical outputs. Equivalence of programs is relative to what can be observed about their computation. For example, the two sorting routines, *binary sort* and *quicksort* [Set89], are equivalent if all I can observe is that the algorithms sort their inputs. However, if I can observe the time taken to sort inputs by *binary sort* and *quicksort*, then they will not be equivalent.

Understanding observational equivalence of programs is of fundamental importance for a variety of reasons. For example, the compiler for SML of New Jersey first transforms programs into a continuation-passing-style (CPS) intermediate language, and then performs various transformations on the program in the intermediate language to improve its run-time performance [App92]. Let us assume that SML programs and their translations to CPS language evaluate to the same value. Still, there is a question regarding the transformations done on the CPS programs — what is the relationship between the original CPS program and the transformed CPS program? Unfortunately, [App92] does not answer this question in a formally precise manner. Suppose I take a program P in the CPS language and transform it to Q following [App92]. What is the relationship between the observations I can make of P and Q? If the observable properties of P and Q are identical, then up to our definition of observations the transformation maintains the essential nature of the program. However, if the observable properties of P and Q are not identical, then there will be situations in which the observable results produced by Q will differ from the ones produced by P. Such a transformation can only be justified if Q is observationally equivalent to P in the environment in which Q is used. *The claim is that observational equivalence provides a framework to study transformations performed by compilers.* Along similar lines, one can want to change pieces of existing programs as better algorithms and implementations are developed. Observational equivalence provides a framework to verify whether two programs can be exchanged.

The study of observational equivalence for functional languages with state has been particularly difficult [MS88, OT93, SF92, MT92, Sie93, OT92]. In specifying Λ_{vs}, I have placed the evaluator for Λ_{vs} within the rich proof theory of FORUM. I use the meta-theory of FORUM to analyze observational equivalence. I first define observational equivalence for Λ_{vs} programs with respect to the natural semantics and the specification in FORUM, $\mathcal{E}_{vs}$. Next, using a theorem in [MT92], I prove that the two definitions are equivalent. I prove that reduction preserves observational equivalence. This is the basis for the equational theory for Λ_{vs} in [SF92]. I also prove the observational equivalence for some of the examples in [MS88, OT92]. One of the main focuses of my future work will be to analyze proof-theoretic properties of the transformations required for proving observational equivalences.

4.1 Defining Observational Equivalence

In this section, I define two notions of observational equivalence. Firstly, I define when two Λ_{vs} (defined in figure 3.8) programs are observationally equivalent with respect to their natural semantics specification (defined in section 3.2). Secondly, I define when two

Λ_{vs} programs are observationally equivalent with respect to $\mathcal{E}_{vs}$, the specification of Λ_{vs} in FORUM, defined in section 3.2. Next, I prove that the two notions of observational equivalence are identical. This result lets me study the observational equivalence for Λ_{vs} programs in FORUM — the main aim of this chapter.

The basic idea of observational equivalence is that one places a program in a *context*, and then *observes* its behavior. Two programs are observationally equivalent if and only if in all contexts the observable behavior remains the same. There are two key concepts here — *context* and *observable behavior*. Suppose I have two programs, M and N, in Λ_{vs}. *In what contexts can I place M and N?* If I think in terms of program transformations, then M is a part of a larger Λ_{vs} program, and I am replacing it with N — *the contexts must come from the syntax of* Λ_{vs}. Following the argument, a context would be a Λ_{vs} program with one of its pieces missing. The idea would then be that programs M and N have identical observational behavior with respect to contexts of the given language, in this case Λ_{vs}.

What are the observable properties of a program? Many observations can be made about the computation of a program in a context. Some of the many observable properties of interest are — whether the evaluation of programs terminate, whether the programs evaluate to the same answer, whether the programs yield the same answer with identical execution time, whether the programs create identical number of new memory locations, etc. The most primitive of the choices listed above is observing whether the programs evaluate to the same answer. As the evaluator for the language is deterministic, observing whether programs evaluate to the same answer yields the same relation as observing whether two programs terminate. The other choices for observations are more refined versions of these two primitive observations. If I transform M to N, the minimal property that I would want of the resulting programs is that one terminates if and only if the other does — guaranteeing the safety of the transformation. *I take termination of the evaluation of programs as the observation I make.*

Contexts, ranged over by C, are defined in figure 4.1. I use $\circ$ to denote the place in the context where the program will be placed. For example, $\lambda x.\circ$ is a context. Note that

$$
\begin{array}{lll}
C ::= & \mid \circ & (\mathit{Contexts}_{vs}) \\
 & \mid x & x \in \mathit{Vars} \\
 & \mid n & n \in \mathcal{Z} \\
 & \mid b & b \in \mathcal{B} \\
 & \mid (\lambda x.\, C) & \\
 & \mid \bullet & \\
 & \mid (f\ C\ C) & f \in \mathcal{O} \\
 & \mid (C\ C) & \\
 & \mid (\mathsf{if}\ C\ C\ C) & \\
 & \mid (\mathsf{let\ val}\ x = C\ \mathsf{in}\ C) & \\
 & \mid (\mathsf{let\ fun}\ f\ x = C\ \mathsf{in}\ C) &
\end{array}
$$

Figure 4.1: Contexts in Λ_{vs}

placing a term, M, in the holes in a context, C, written as $C[M]$, may cause binding of free variables in the term. For example, if I place x in the hole in $\lambda x.\circ$, then I get $\lambda x.\, x$ — *the free variable x gets captured by the context.* I now define when a program terminates in a state with respect to natural semantics.

Definition 4.1 [Termination of programs in Λ_{vs} in natural semantics] Let $M \in \Lambda_{vs}$ and S_0 be a state such that $\mathsf{close}(S_0, M)$. M in the state S_0 terminates, written as $(M, S_0)\downarrow_{ns}$, if there exists a value V and a state S_1 such that $\langle M, S_0\rangle \Downarrow \langle V, S_1\rangle$. ■

Observational equivalence using program contexts was first defined by [Mor68]. It has been studied extensively for call-by-name and call-by-value Λ-calculus, and its extensions with state [Abr87, AO89, Abr90, Hoa69, MT92, MS88, OT93, Plo76, PS93, SF92]. The definition of observational equivalence for Λ_{vs} is an extension to the definition for Λ_v [Mor68] along the lines of [MT92].

Definition 4.2 [Observational Equivalence with natural semantics] Let M and N be two Λ_{vs} terms, and $\emptyset$ be the empty state. M is said to be observationally equivalent to N with respect to the natural semantics, written as $M \cong_{ns} N$, if
$\forall C \in \mathit{Contexts}_{vs}$ such that $C[M]$ and $C[N]$ are closed terms, $(C[M], \emptyset)\downarrow_{ns}$ if and only if $(C[N], \emptyset)\downarrow_{ns}$. ■

To determine whether two programs are equivalent, I have to check for their termination in all contexts in $Contexts_{vs}$. This obviously becomes a very difficult problem because one has no control over what the arbitrary contexts might do. [MT92] provides an alternate definition of observational equivalence in which they are able to reduce the contexts to $EvCont_{vs}$, the evaluation contexts. I use a slight variation of the definition of $\cong^{ciu}$ in [MT92].

Definition 4.3 [$\cong^{ciu}$] Let M and N be two Λ_{vs} terms. M is said to be observationally equivalent to N with respect to natural semantics, written as $M \cong^{ciu} N$, if
for all $E \in EvCont_{vs}$ and for all states S, such that $\mathsf{close}(S, E[M])$ and $\mathsf{close}(S, E[N])$, $(E[M], S)\downarrow_{ns}$ if and only if $(E[N], S)\downarrow_{ns}$. ■

Going from arbitrary contexts to evaluation contexts, one loses the capability to bind variables using λ and state. Since we are only evaluating programs which are closed with respect to the λ bound variables, the main concern is the binding of assignable variables. Consider the two programs $(\mathsf{asg}\ l\ 5; 1)$ and $(\mathsf{asg}\ l\ 7; 1)$, where $M; N$ is syntactic sugar for $((\lambda d.\ N)\ M)$, $d \notin FV(N)$. I evaluate the programs in a state which maps l to 0. Clearly, the two programs are not observationally equivalent with respect to the definition for $\cong_{ns}$. However, if in the definition of $\cong^{ciu}$ I had insisted on E being closed, then the two programs would have been equivalent! The point is that $E[M]$ may have free variables which are defined in the state in which $E[M]$ will be evaluated. The testing of the program is done by not only placing it in different evaluation contexts, but also by altering the state in which it is evaluated. The following theorem showing the equivalence of $\cong_{ns}$ and $\cong^{ciu}$ was proved in [MT92].

Theorem 4.4 (Theorem ciu, [MT92]) *Let M and N be two Λ_{vs} terms.*
$M \cong_{ns} N$ if and only if $M \cong^{ciu} M$.

I translate the definition of $\cong^{ciu}$ into FORUM. In order to do this, I have to generalize the definition of eval_{vs}, because the initial continuation instead of being K, as in the definition

$$
\begin{array}{rcl}
\mathcal{C}_{vs}([\,], K) & = & K \\
\mathcal{C}_{vs}(E[f\,[\,]\,M], K) & = & \lambda v.\,(\mathsf{eval}\;\mathcal{H}_{vs}(M)\;\lambda u.\,K'\,\underline{f\,v\,u}) \\
\mathcal{C}_{vs}(E[f\,V\,[\,]], K) & = & \lambda v.\,K'\,\underline{f\,\phi_{vs}(V)\,v} \\
\mathcal{C}_{vs}(E[[\,]\,M], K) & = & \lambda v.\,(\mathsf{eval}\;\mathcal{H}_{vs}(M)\;\lambda u.\,(\mathsf{apply}\;v\;u\;K')) \\
\mathcal{C}_{vs}(E[V\,[\,]], K) & = & \lambda v.\,(\mathsf{apply}\;\phi_{vs}(V)\;v\;K') \\
\mathcal{C}_{vs}(E[\mathsf{if}\,[\,]\,M\;N], K) & = & \lambda v.\,(\mathsf{eval}\;(\mathsf{ifbr}\;v\;\mathcal{H}_{vs}(M)\;\mathcal{H}_{vs}(N))\;K') \\
\mathcal{C}_{vs}(E[\mathsf{let\;val}\;x = [\,]\;\mathsf{in}\;M], K) & = & \lambda v.\,(\mathsf{eval}\;\mathcal{H}_{vs}(M)[x := v]\;K') \\
\mathcal{C}_{vs}(E[\mathsf{cell}\,[\,]], K) & = & \lambda v.\,\forall P, l.\,\mathsf{getC}(P,l) \Rightarrow \mathsf{setC}(P,l) \Rightarrow [(K'\,l)\,\wp\,(P\,v)] \\
\mathcal{C}_{vs}(E[\mathsf{read}\,[\,]], K) & = & \lambda v.\,(\mathsf{get}\;v\;K') \\
\mathcal{C}_{vs}(E[\mathsf{write}\,[\,]\,M], K) & = & \lambda v.\,(\mathsf{eval}\;\mathcal{H}_{vs}(M)\;\lambda u.\,(\mathsf{set}\;v\;u\;K')) \\
\mathcal{C}_{vs}(E[\mathsf{write}\;V\,[\,]], K) & = & \lambda u.\,(\mathsf{set}\;\phi_{vs}(V)\;u\;K')
\end{array}
$$

where $$K' =_{def} \mathcal{C}_{vs}(E, K : \mathsf{vl} \to o)$$

Figure 4.2: $\mathcal{C}_{vs}$, translation of $EvCont_{vs}$ to FORUM terms of type $\mathsf{vl} \to o$

of eval_{vs}, is now specified by E. I first define $\mathcal{C}_{vs}$, the translation of E into FORUM in figure 4.2. The definition of eval_{vs} needs to be changed.

Definition 4.5 [Evaluation in FORUM given an initial continuation E] Let $M \in \Lambda_{vs}$, $E \in EvCont_{vs}$ and S_0 be a state such that $\mathsf{close}(S_0, E[M])$. M with S_0 in the initial continuation E evaluates to V with S_1, written as $\mathsf{eval}_s(E, M, S_0, V, S_1)$, if

$$\Sigma_{vs} : \mathcal{E}_{vs}\,;\; \longrightarrow \forall K : \mathsf{vl} \to o.\; {S_1}^o(K\,V) \multimap {S_0}^o(\mathsf{eval}\;\mathcal{H}_{vs}(M)\;\mathcal{C}_{vs}(E, K))$$

is provable in FORUM. ■

It is quite obvious that eval_{vs} will be true of $E[M]$ exactly when eval_s is true of M in the continuation E. The following lemma states the relationship precisely, and follows from an easy induction on the structure of E and the definition of $\mathcal{C}_{vs}$.

Lemma 4.6 *Let $M \in \Lambda_{vs}$, $E \in EvCont_{vs}$, $V \in Values_{vs}$, S_1 be a state, and S_0 be a state such that* $\mathsf{close}(S_0, E[M])$.
$\mathsf{eval}_{vs}(E[M], S_0, V, S_1)$ *if and only if* $\mathsf{eval}_s(E, M, S_0, V, S_1)$

I now define the termination property, $\downarrow_f$, for programs with respect to $\mathcal{E}_{vs}$ using the notion of evaluation defined by eval_s. Next, I define when two Λ_{vs} programs are observationally equivalent on the basis of their evaluation in FORUM.

Definition 4.7 [Termination of programs in Λ_{vs} in FORUM] Let $M \in \Lambda_{vs}$, $E \in \mathit{EvCont}_{vs}$, and S_0 be a state such that $\mathsf{close}(S_0, E[M])$. M in the state S_0 with the initial continuation E terminates, written as $(E, M, S_0)\downarrow_f$, if there exists a value V, and a state S_1 such that $\mathsf{eval}_s(E, M, S_0, V, S_1)$. ■

Definition 4.8 [Observational Equivalence with $\mathcal{E}_{vs}$] Let M and N be two Λ_{vs} terms. M is said to be observationally equivalent to N with respect to $\mathcal{E}_{vs}$, written as $M \cong_f N$, if for all $E \in \mathit{EvCont}_{vs}$ and for all states S, such that $\mathsf{close}(S, E[M])$ and $\mathsf{close}(S, E[N])$, $(E, M, S)\downarrow_f$ if and only if $(E, N, S)\downarrow_f$. ■

In order to use the translation in FORUM to prove observational equivalence, I first need to prove that $\cong_f$ defines the same relation as $\cong^{ciu}$.

Theorem 4.9 ($\cong^{ciu}$ and $\cong_f$ coincide) *Let M and N be two Λ_{vs} terms.*
$M \cong^{ciu} N$ if and only if $M \cong_f N$.

Proof: To prove the above theorem, it is enough to prove that $(E, M, S_0)\downarrow_f$ if and only if $(E[M], S_0)\downarrow_{ns}$, for any state S_0 and $E \in \mathit{EvCont}_{vs}$, such that $\mathsf{close}(S_0, E[M])$ and $\mathsf{close}(S_0, E[N])$.

Unraveling definitions of $\downarrow_f$ and $\downarrow_{ns}$, all I need to prove is that if for some $V \in \mathit{Values}_{vs}$ and state S_1, $\mathsf{eval}_s(E, M, S_0, V, S_1)$ then $\langle E[M], S_0\rangle \Downarrow \langle V, S_1\rangle$. Conversely, if for some $V \in \mathit{Values}_{vs}$ and state S_1, $\langle E[M], S_0\rangle \Downarrow \langle V, S_1\rangle$ then $\mathsf{eval}_s(E, M, S_0, V, S_1)$.

By the Correspondence theorem 3.8 in section 3.2

$$\mathsf{eval}_{vs}(E[M], S_0, V, S_1) \textit{ if and only if } \langle E[M], S_0\rangle \Downarrow \langle V, S_1\rangle.$$

Further note that by lemma 4.6

$$\mathsf{eval}_{vs}(E[M], S_0, V, S_1) \text{ \textit{if and only if} } \mathsf{eval}_s(E, M, S_0, V, S_1).$$

Using these facts the proof is completed. ■

Now suppose I want to prove that two programs M and N are observationally equivalent. Suppose, for some $E \in EvCont_{vs}$, states S_0 and S_1, and $V \in Values_{vs}$ such that $FV(E[M]) \subset$ $\mathsf{dom}(S)$, $\mathsf{eval}_s(E, M, S_0, V, S_1)$ is true. I look at the resulting proof tree in FORUM and transform it to a proof tree for $\mathsf{eval}_s(E, N, S_0, V, S_1)$. If I can exhibit such transformation to and fro, then I have established that M is observationally equivalent to N. *The problem of determining whether two programs are observationally equivalent has been reduced to specifying proof transformations.*

4.2 Reduction in Λ_{vs} preserves Observational Equivalence

In this section, I use $\mathcal{E}_{vs}$ to prove that if a program reduces to another program, then the two programs are observationally equivalent – the basis for the equational theory in [FH92]. eval_s defines the evaluation of a term M in the state S, and continuation E to a value V and state S. In this sense, eval_s is not specifying reductions, rather, it is specifying complete evaluations. However, using lemma 4.6 it can be easily proved that

$$\mathsf{eval}_s(E, M, S_0, V, S_1) \text{ \textit{if and only if} } \mathsf{eval}_s([\,], E[M], S_0, V, S_1).$$

Using this basic intuition, I define when M reduces to N.

Definition 4.10 [Reduction in Λ_{vs}] Let $M, N \in \Lambda_{vs}$, S_1 be state, and S_0 be a state such that $\mathsf{close}(S_0, M)$. M in state S_0 evaluates to N in state S_1, written as $\mathsf{red}_s(M, S_0, N, S_1)$, if

$\Sigma_{vs} : \mathcal{E}_{vs}\, ;\ \longrightarrow \forall K : \mathsf{vl} \rightarrow o.\ S_1{}^o(\mathsf{eval}\ \mathcal{H}_{vs}(N)\ K) \multimap S_0{}^o(\mathsf{eval}\ \mathcal{H}_{vs}(M)\ K)$ is provable in FORUM. ■

I would like to prove that reduction preserves observational equivalence, *i.e.* if $\mathsf{red}_s(M, S_0, N, S_1)$ then $M \cong_f N$. Unfortunately, as stated my claim would be false. Following is a counter-example.

Example 4.11 Let $M =_{def} \mathsf{ref}\ 0$, $S_0 =_{def} \emptyset$, $N =_{def} l$, and $S_1 =_{def} \langle l, 0\rangle$. Clearly $\mathsf{red}_s(M, S_0, N, S_1)$ is true.

However, $\mathsf{ref}\ 0$ is not observationally equivalent to l. To distinguish the two terms, take $E = \mathsf{deref}\ [\,]$ and $S = \langle l, P\rangle$, where P is a divergent program. ■

The problem is that evaluation of a program may create new memory cells and change the existing state. However, the statement $M \cong_f N$ throws away this information. In the above example l is defined in the state S_1, but this information was not used in the attempted proof of $\mathsf{ref}\ 0 \cong_f l$. On the other hand, $(\mathsf{ref}\ 0)$ is observationally equivalent to l with respect to all states which map l to 0. Suppose we had a way of representing state in the syntax of Λ_{vs}, then the situation can be repaired. I would change the succedent of my claim to $M' \cong_f N'$, where M' and N' are Λ_{vs} terms, such that M' incorporates the state S_0 and M, while N' incorporates the state S_1 and N. I will prove the new statement of my claim below. I define the translation of a state into the syntax of Λ_{vs} [MT92, SF92]. $(M; N)$ is syntactic sugar for $(\lambda d.\, N)\, M$, $d \notin FV(N)$.

Definition 4.12 Given a state S, I define a Λ_{vs} term S^+, the encoding of state S in Λ_{vs}.

$$S^+ =_{def} ((\lambda x_1, \ldots, x_n, y.\mathsf{asg}\ x_1\ V_1; \ldots ; \mathsf{asg}\ x_1\ V_1; y)(\mathsf{ref}\ 1) \ldots (\mathsf{ref}\ n))$$

Where

- $l_S = x_1, \ldots, x_n$.

- y is distinct from all x_i, $i \in [1, n]$.
- $S(x_i) = V_i$, $i \in [1, n]$.

■

The proof of the following lemma is immediate from the construction of S^+ for a state S.

Lemma 4.13 *Let $N \in \Lambda_{vs}$, and S a state such that* $\mathsf{close}(S, N)$.
$\mathsf{red}_s((S^+ N), \emptyset, N, S)$ *is true.*

Theorem 4.14 *Let M be a redex, $N \in \Lambda_{vs}$, S_1 be a state, and S_0 a state such that* $\mathsf{close}(S_0, M)$.
If $\mathsf{red}_s(M, S_0, N, S_1)$ *then* $({S_0}^+ M) \cong_f ({S_1}^+ N)$.

Proof: Assume given $M, N \in \Lambda_{vs}$, S_0 a state such that $\mathsf{close}(S_0, M)$, and $\mathsf{red}_s(M, S_0, N, S_1)$ is true. Let $M' =_{def} ({S_0}^+ M)$ and $N' =_{def} ({S_1}^+ N)$. Unraveling the definition of $M' \cong_f N'$, I have to prove that for any arbitrary state S and $E \in \mathit{EvCont}_{vs}$, such that $\mathsf{close}(S, E[M'])$ and $\mathsf{close}(S, E[N'])$, $(E, M', S)\downarrow_f$ if and only if $(E, N', S)\downarrow_f$.

Going from right to left, I have to prove that if there exists $V \in \mathit{Values}_{vs}$, and a state S' such that $\mathsf{eval}_s(E_1, N', S, V, S')$, then $\mathsf{eval}_s(E_1, M', S, V, S')$.

Assume $\mathsf{eval}_s(E_1, N', S, V, S')$. I have to construct a proof for $\mathsf{eval}_s(E_1, M', S, V, S')$. $\mathcal{E}_{vs}$ is in the intuitionistic part and Σ_{vs}, K is in the signature of all the sequents shown in all the proofs that I construct. I start by constructing a proof below, called σ_1. In σ_1, δ_1 is obtained by unfolding the definition for $\mathsf{red}_s(M, S_0, N, S_1)$ and lemma 4.13 gives me δ_2. Let $M_1 =_{def} \mathcal{H}_{vs}(M)$, $N_1 =_{def} \mathcal{H}_{vs}(N)$, $M_1' =_{def} \mathcal{H}_{vs}(M')$, $N_1' =_{def} \mathcal{H}_{vs}(N')$, $K' =_{def} \mathcal{C}_{vs}(E, K)$, and $V_1 =_{def} \phi_{vs}(V)$.

$$\dfrac{{S_1}^o(\textsf{eval}\ N_1\ K) \overset{\delta_1}{\longrightarrow} {S_0}^o(\textsf{eval}\ M_1\ K) \quad {S_0}^o(\textsf{eval}\ M_1\ K) \overset{\delta_2}{\longrightarrow} (\textsf{eval}\ M_1'\ K)}{{S_1}^o(\textsf{eval}\ N_1\ K) \longrightarrow {S_0}^o(\textsf{eval}\ M_1'\ K)}\ \mathit{CutS}$$

By definition of $\textsf{eval}_s$, and N' I must have the following proof in FORUM.

$$\dfrac{\dfrac{\dfrac{\Sigma_S, \Sigma_{S_1}, K : \textsf{CL}_S\,;\, S'^o(K\,V_1) \overset{\gamma_1}{\longrightarrow} (\textsf{eval}\ N_1\ K') \wp \Gamma_S \wp \Gamma_{S_1}}{\vdots\ \mathit{backchain\ on}\ \textsf{ref}\ \mathit{clause}, \forall R, \Rightarrow R}}{\Sigma_S, K : \textsf{CL}_S\,;\, S'^o(K\,V_1) \longrightarrow (\textsf{eval}\ N_1'\ K') \wp \Gamma_S}}{\dfrac{\vdots\ \forall R, \Rightarrow R}{\longrightarrow \forall K : \textsf{vl} \to o.\, S'^o(K\,V_1) \multimap S^o(\textsf{eval}\ N_1'\ K')}}$$

Using γ_1 in the above proof, I construct σ_2 below.

$$\dfrac{\dfrac{\Sigma_S, \Sigma_{S_1}, K : \textsf{CL}_S\,;\, S'^o(K\,V_1) \overset{\gamma_1}{\longrightarrow} (\textsf{eval}\ N_1\ K') \wp \Gamma_S \wp \Gamma_{S_1}}{\vdots\ , \Rightarrow R, \forall R}}{\Sigma_S, K : \textsf{CL}_S\,;\, S'^o(K\,V_1) \longrightarrow S_1^o(\textsf{eval}\ N_1\ K') \wp \Gamma_S}$$

Using σ_2, I construct the required proof below. To keep the proof readable, I do not write Σ_S, K in the signature, and $\textsf{CL}_S$ in the intuitionistic part of some of the sequents.

$$\dfrac{\dfrac{S'^o(K\,V_1) \overset{\sigma_2}{\longrightarrow} (\textsf{eval}\ N_1\ K') \wp \Gamma_S \quad {S_1}^o(\textsf{eval}\ N_1\ K') \overset{\sigma_4}{\longrightarrow} (\textsf{eval}\ M_1\ K')}{\Sigma_S, K : \textsf{CL}_S\,;\, S'^o(K\,V_1) \longrightarrow (\textsf{eval}\ M_1'\ K') \wp \Gamma_S}\ \mathit{CutS}}{\dfrac{\vdots\ \forall R, \Rightarrow R}{\longrightarrow \forall K : \textsf{vl} \to o.\, S'^o(K\,V_1) \multimap S^o(\textsf{eval}\ M_1'\ K')}}$$

To complete the construction, I build σ_4 below. σ_3 is obtained from σ_1 by inflating the signature and the intuitionistic parts of the sequents in the proof.

$$\dfrac{K' \text{ is a } \Sigma_S \text{ term} \quad \Sigma_S, K : \textsf{CL}_S\,;\, {S_1}^o(\textsf{eval}\ N_1\ K_1) \overset{\sigma_3}{\longrightarrow} {S_0}^o(\textsf{eval}\ M_1\ K_1)}{\Sigma_S, K : \textsf{CL}_S\,;\, {S_1}^o(\textsf{eval}\ N_1\ K') \longrightarrow (\textsf{eval}\ M_1\ K')}\ \mathit{CutS}$$

Going from left to right, I use the fact the $\mathcal{E}_{vs}$ is deterministic, because there is exactly one clause for every term constructor. Hence, if $\mathsf{red}_s(M, S_0, N, S_1)$, then every evaluation of M that evaluates the term beyond N must pass through the reduction of M to N. Assuming $\mathsf{eval}_s(E_1, M', S, V, S')$, I have to construct a proof of $\mathsf{eval}_s(E_1, N', S, V, S')$. The proof in $\mathsf{eval}_s(E_1, M', S, V, S')$ must have the following shape. To keep the proof readable, I do not write the signature and the intuitionistic part of the sequent in all the sequents in the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\delta}{\Sigma_S, \Sigma_{S_1} : \mathsf{CL}_S, \mathsf{CL}_{S_0} \,;\, S'^o(K\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ K') \wp \Gamma_S \wp \Gamma_{S_1}}\ \vdots\ \textit{reduction of } M_1 \textit{ in } S_0 \textit{ to } N_1 \textit{ in } S_1}{S'^o(K\,V_1) \longrightarrow (\mathsf{eval}\ M_1\ K') \wp \Gamma_S \wp \Gamma_{S_0}}\ \vdots\ \textit{backchain on } \mathsf{ref} \textit{ clause}, \forall R, \Rightarrow R}{S'^o(K\,V_1) \longrightarrow (\mathsf{eval}\ M_1'\ K') \wp \Gamma_S}\ \vdots\ \forall R, \Rightarrow R}{S'^o(K\,V_1) \longrightarrow S^o(\mathsf{eval}\ M_1'\ K')}$$

Using δ I construct the required proof below.

$$\dfrac{\dfrac{\dfrac{\delta}{\Sigma_S, \Sigma_{S_1} : \mathsf{CL}_S, \mathsf{CL}_{S_0} \,;\, S'^o(K\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ K') \wp \Gamma_S \wp \Gamma_{S_1}}\ \vdots\ \textit{backchain on } \mathsf{ref} \textit{ clause}, \forall R, \Rightarrow R}{\Sigma_S : \mathsf{CL}_S \,;\, S'^o(K\,V_1) \longrightarrow (\mathsf{eval}\ N_1'\ K') \wp \Gamma_S}\ \vdots\ \forall R, \Rightarrow R}{S'^o(K\,V_1) \longrightarrow {S_0}^o\,(\mathsf{eval}\ N_1'\ K')}$$

■

The above proof highlights some key aspects $\mathcal{E}_{vs}$. If M reduces to N, then N', the translation of N, entails M', the translation of M. Hence, whenever N' evaluates to a value, so will M'. The proof going from right to left uses this fact and *CutS* to construct the required proof. Now, N' entails M'. This does not necessarily imply that an evaluation of M' has to have N' as an intermediate state. Since $\mathcal{E}_{vs}$ is deterministic, *i.e.* for every term construct of type tm there is exactly one clause, if M' evaluates to a value, and N' entails M', it necessarily follows that the evaluation has N' in some intermediate state. This observation yields the proof going from left to right in the above theorem.

4.3 Observational Equivalence proofs in FORUM

In this section I present the Meyer-Sieber examples [MS88] in the UML, and prove the desired equivalences, interpreting equivalence as $\cong_f$. I have converted the Algol-like notation of [MS88] into UML syntax following [MT92]. State introduces new nuances into the programming language, and correspondingly into any theory which tries to study the equivalence of programs with state. The idea behind the examples was to highlight some of the novel issues that come up when state is added to a higher order language. Let Ω be a divergent program in UML.

Example 4.15 [Example 1] Let $M \in \Lambda_{vs}$, and $M_1 =_{def} \mathsf{let\ val}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ M$, $x \notin FV(M)$.

$$M \cong_f M_1$$

The intuitive justification is simple. As $x \notin FV(M)$, the creation of x has no effect on the behavior of M. ■

The proof for this example follows from the following more general statement of the problem. The lemma is proved by a straightforward induction on the height of the evaluations of $(S_3^+\ M)$ and M in FORUM. The essential point is that the evaluation of $(S_3^+\ M)$ creates new locations by picking new eigen-variables in FORUM. Hence, if $(S_3^+\ M)$ is placed in an evaluation context E, then E cannot access the newly created locations. The computations of $(S_3^+\ M)$ and M are identical except for this difference.

Lemma 4.16 (Elimination of inaccessible cells) *Let $M \in \Lambda_{vs}$, $L =_{def} l_1, \ldots, l_n$, and S_1 be a state such that :*

- $\mathsf{close}(S_1, M)$.
- $L \subset \mathsf{dom}(S_1)$.
- *for all* $l \in L$, $l \notin FV(M)$.
- *if* $l \in FV(M)$, *then for all* $l' \in L$, $l' \notin FV(S_1(l))$.
- S_3 *is the restriction of* S_1 *to* L.

$$(S_3^+\, M) \cong_f M$$

I first point out some equivalences for which I do not need induction on the height of proofs. The proofs in FORUM for these terms are permuted to each other.

Remark 4.17 *Let* $M, N, P\Lambda_{vs}$, *and* $V \in \Lambda_{vs}$.

1. $\mathsf{let\ val}\ x = V\ \mathsf{in}\ M \cong_f (\lambda x.\, M)V$.
2. $E[M] \cong_f \mathsf{let\ val}\ x = M\ \mathsf{in}\ E[x]$, $x \notin FV(E)$.
3. $E[\mathsf{let\ val}\ x = M\ \mathsf{in}\ N] \cong_f \mathsf{let\ val}\ x = M\ \mathsf{in}\ E[N]$, $x \notin FV(E)$.
4. $\mathsf{let\ val}\ x = \mathsf{ref}\ M\ \mathsf{in}\ E[\bullet] \cong_f \mathsf{let\ val}\ x = \mathsf{ref}\ V\ \mathsf{in}\ E[\mathsf{asg}\ x\ M]$, $x \notin FV(M)$.
5. $\mathsf{let\ val}\ x = \mathsf{ref}\ P\ \mathsf{in}\ ((\lambda z.\, M)\ N) \cong_f ((\mathsf{let\ val}\ x = \mathsf{ref}\ P\ \mathsf{in}\ \lambda z.\, M)\, N)$, $x \notin FV(N)$
6. $\mathsf{let\ val}\ x = \mathsf{ref}\ V\ \mathsf{in}\ ((\lambda z.\, M)\ N) \cong_f ((\lambda z.\, \mathsf{let\ val}\ x = \mathsf{ref}\ V\ \mathsf{in}\ M)\, N)$, $x \notin FV(N)$, $z \notin FV(M) \cup FV(V)$

Equivalence 1 is obvious from the interpretation of letval. Equivalence 2 and 3 follow from lemma 4.6. For equivalence 4, observe that x *is inaccessible* from M, thus evaluating M with or without x declared in the environment makes no difference. Equivalences 5 and 6 result from the permutation of proofs because of information about variable occurrences.

Example 4.18 [Example 2] $M \in \Lambda_{vs}$.

- $M_1 \ =_{def} \ \Omega$.

- $M_2 \ =_{def}$ $\mathsf{let\ val}\ x = (\mathsf{ref}\ \bullet)\ \mathsf{in}\ ((\mathsf{asg}\ x\ \mathsf{true})\,;\,(M\,\bullet)\,;\,\mathsf{if}\ (\mathsf{deref}\ x)\ \Omega\ 1)$, $x \notin FV(M)$.

■

Lemma 4.19 $M_{2.1} \cong_f M_{2.2}$

Proof: The strategy is to show that for any $E \in EvCont_{vs}$, and any state S such that $\mathsf{close}(S, M_2)$, $(E, M_2, S)\downarrow_f$ is not true.

On the contrary, suppose that $(E, M_2, S_0)\downarrow_f$ is true for some $E_0 \in EvCont_{vs}$, and state S_0 such that $\mathsf{close}(S_0, E_0[M_2])$. Then there exists a value V and a state S_1 such that $\mathsf{eval}_s(E_0, M_2, S_0, V, S_1)$ is true. Now by the equivalence in remark 4.17,

$$M_2 \cong_f ((M\,\bullet)\,;\,\mathsf{let\ val}\ x = (\mathsf{ref}\ 0)\ \mathsf{in}\ (\mathsf{if}\ (\mathsf{deref}\ x)\ \Omega\ 1)).$$

- $V' \ =_{def} \ \phi_{vs}(V)$.
- $M' \ =_{def} \ \mathcal{H}_{vs}(M)$.
- $\Omega' \ =_{def} \ \mathcal{H}_{vs}(\Omega)$.
- $Q \ =_{def} \ (\mathsf{let\ val}\ x = (\mathsf{ref}\ 0)\ \mathsf{in}\ (\mathsf{if}\ (\mathsf{deref}\ x)\ \Omega\ 1))$, $Q' \ =_{def} \ \mathcal{H}_{vs}(Q)$.
- $N \ =_{def} \ ((M\,\bullet)\,;Q)$, $N' \ =_{def} \ \mathcal{H}_{vs}(N)$.
- $C_1 \ =_{def} \ \mathcal{C}_{vs}(E\,K)$.
- $C_2 \ =_{def} \ \lambda u.\,(\mathsf{apply}\ (\mathsf{abs}\ \lambda d.\,Q')\ u\ C_1)$, $d \notin FV(Q)$.

By definition of $\cong_f$ I have a proof δ of

$$\Sigma_{vs}, K, \Sigma_{S_0} : \mathcal{E}_{vs}, \mathsf{CL}_{S_0}\,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\ N'\ C_1)\wp\Gamma_S$$

in FORUM.

The last rules of δ must be as shown below.

$$\dfrac{\dfrac{\delta_1}{\Sigma_{vs}, K, \Sigma_{S_0} : \mathcal{E}_{vs}, \mathsf{CL}_{S_0} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\,M'\,\bullet)\,C_2) \wp \Gamma_{S_0}}}{\dfrac{\vdots}{\Sigma_{vs}, K, \Sigma_{S_0} : \mathcal{E}_{vs}, \mathsf{CL}_{S_0} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\,N'\,C_1) \wp \Gamma_{S_0}}}$$

Suppose $\langle (M\,\bullet), S_0\rangle$ diverges then there cannot exist a δ_1 because of theorem 3.8. There is a contradiction, hence I am done.

Suppose $\langle (M\,\bullet), S_0\rangle \Downarrow \langle U, S_2\rangle$. Hence, by theorem 3.8 I would get δ_1 as shown below. Let $U' =_{def} \phi_{vs}(U)$, $S_3 =_{def} S_2[l \mapsto 0]$, $l \notin \mathsf{dom}(S_2)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\delta_2}{\Sigma_{vs}, K, \Sigma_{S_3} : \mathcal{E}_{vs}, \mathsf{CL}_{S_3} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\,\Omega'\,C_1) \wp \Gamma_{S_3}}}{\dfrac{\vdots}{\Sigma_{vs}, K, \Sigma_{S_2} : \mathcal{E}_{vs}, \mathsf{CL}_{S_2} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\,Q'\,C_1) \wp \Gamma_{S_2}}}}{\Sigma_{vs}, K, \Sigma_{S_2} : \mathcal{E}_{vs}, \mathsf{CL}_{S_2} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{apply}\,(\mathsf{abs}\,\lambda d.\,Q')\,U'\,C_1) \wp \Gamma_{S_2}}}{\dfrac{\vdots}{\Sigma_{vs}, K, \Sigma_{S_0} : \mathcal{E}_{vs}, \mathsf{CL}_{S_0} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\,M'\,\bullet)\,C_2) \wp \Gamma_{S_0}}}$$

Using theorem 3.8 and the fact that Ω is divergent I get that δ_2 cannot exist in FORUM. ■

Example 4.20 [Example 3] Let $M \in \Lambda_{vs}$.

- $M_{3.1} =_{def}$ let val $x = (\mathsf{ref}\ 0)$ in (let val $y = (\mathsf{ref}\ 0)$ in M).
- $M_{3.2} =_{def}$ let val $y = (\mathsf{ref}\ 0)$ in (let val $x = (\mathsf{ref}\ 0)$ in M).

■

Lemma 4.21 $M_{3.1} \cong_f M_{3.2}$

Proof: The idea is to show that given a proof for the evaluation of $M_{3.1}$ I can transform it to an evaluation of $M_{3.2}$, and vice versa.

Let E be an arbitrary evaluation context and S_0 any state such that $\mathsf{close}(S_0, E[M_{3.1}])$. Suppose $\langle E[M_{3.1}], S_0\rangle \Downarrow \langle V, S_1\rangle$, for some value V and state S_1. Let $V' =_{def} \phi_{vs}(V)$, $C_1 =_{def} \mathcal{C}_{vs}(E, K)$, $N_1' =_{def} \mathcal{H}_{vs}(M_{3.1})$, $N_2' =_{def} \mathcal{H}_{vs}(M_{3.2})$.

By theorem 3.8, I have a proof in FORUM, δ_1, of

$$\Sigma_{vs}, \Sigma_{S_0}, K : \mathcal{E}_{vs}, \mathsf{CL}_{S_0} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\ N_1'\ C_1) \wp \Gamma_{S_0}.$$

I need to transform δ_1, into a proof for the evaluation of N_2' in FORUM. The shape for δ_1 is shown below.

$$\cfrac{\cfrac{\delta_2}{\Sigma_{vs}, \Sigma_{S_1}, K : \mathcal{E}_{vs}, \mathsf{CL}_{S_1} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ (\mathsf{app}\ Q'\ l_1)\ l_2)\ C_1) \wp \Gamma_{S_0} \wp (P_1\,0) \wp (P_2\,0)}}{\cfrac{\vdots}{\Sigma_{vs}, \Sigma_{S_0}, K : \mathcal{E}_{vs}, \mathsf{CL}_{S_0} \,;\, S_1^o(K\,V') \longrightarrow (\mathsf{eval}\ N_1'\ C_1) \wp \Gamma_{S_0}}}$$

Using δ_2, I can trivially get the evaluation for $M_{3.2}$ in FORUM. The proof in the other direction follows from a similar argument. ■

The above proof shows the advantage of using eigen-variables to generate names of new assignable variables. The two terms in this example can essentially be renamed to each other. This fact is made precise by the usage of the eigen-variables.

Example 4.22 [Example 4] Let $M \in \Lambda_{vs}$.

- $M_{4.1} =_{def} \Omega$.
- $M_{4.2} =_{def}$ $\mathsf{let\ val}\ x = (\mathsf{ref}\ 0)\ \mathsf{in}$
  $\mathsf{let\ val}\ f = \lambda d.\,(\mathsf{asg}\ x\ (+\ 2\ (\mathsf{deref}\ x)));(\mathsf{deref}\ x)\ \mathsf{in}$
  $(M\ f); (\mathsf{if}\ (=\ 0\,(\mathsf{mod}\,(\mathsf{deref}\ x)\ 2))\ \Omega\ 1)$, $x, f \notin FV(M)$.

■

Lemma 4.23 $M_{4.1} \cong_f M_{4.2}$

Proof: The strategy is to show that $M_{4.2}$ will have no evaluation in FORUM. I only consider the case when $(M\ f)$ converges, as otherwise the argument is trivial. I introduce a new idea in this proof.

The essential point of the example is that access to the local variable x is passed to M only via f. Hence, the content of x can only be incremented by 2, and then read out — no other operation is possible on x. In a way, an abstract data type has been created with the only interface function being f. The if statement checks whether this abstraction was maintained by M or not.

Let

$$F =_{def} \forall K, V, U. [(\mathsf{apply}\ c\ V\ K) \wp (P\, U)] \multimapinv [(K\, \underline{2 * (U/2 + 1)}) \wp (P\, \underline{2 * (U/2 + 1)})].$$

P and c are declared in the signature. F encapsulates the computational behavior of f. Note how it does away with the name of the cell $(P\, U)$, and need for getC and setC clauses for P.

Suppose I have a proof, δ, in FORUM of

$$\Sigma_{vs}, \Sigma_{S_0}, K, c, P, l : \mathcal{E}_{vs}, \mathsf{CL}_{S_0}, F\,;\, S_1^{\circ}(K, V') \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ M'\ c)\ C_1) \wp \Gamma_{S_0} \wp (P\, 0),$$

where $E \in EvCont_{vs}$,

- S_0 is a state such that $\mathsf{close}(S_0, E[M])$.
- $V' =_{def} \phi_{vs}(V)$.
- $M' =_{def} \mathcal{H}_{vs}(M)$.

- $C_1 =_{def} \mathcal{C}_{vs}(E, K)$.

In δ the only interaction for P is via F as there are no getC and setC clauses for P and l in the sequent above. *Thus, it is trivially true that $S_1(l)$ is a multiple of* 2.

I construct δ_1 from δ using *CutS*. Let $F' =_{def} F[c := \mathrm{Add}_2]$, and

$$\mathrm{Add}_2 =_{def} \mathcal{H}_{vs}(\lambda d.\, (\mathsf{asg}\ x\ (+\ 2\ (\mathsf{deref}\ x))); (\mathsf{deref}\ x)).$$

$$\frac{\mathrm{Add}_2 \text{ is a } \Sigma_{vs}, l \text{ term} \quad \overset{\delta}{c, P, l : F\,;\ S_1^o(K, V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\ M'\ c)\ C_1) \wp \Gamma_{S_0} \wp (P\,0)}}{P, l : F'\,;\ S_1^o(K, V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\ M'\ \mathrm{Add}_2)\ C_1) \wp \Gamma_{S_0} \wp (P\,0)}$$

I construct δ_2 below. Let $\mathsf{CL}_l =_{def} \{\mathsf{getC}(P, l), \mathsf{setC}(P, l)\}$. In this proof I have used the equation $2 * (U/2 + 1) = U + 2$, which is true if the division is for real numbers.

$$\dfrac{\dfrac{\dfrac{}{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ (K\,\underline{U+2}) \wp (P\,\underline{U+2}) \longrightarrow (K\,\underline{U+2}) \wp (P\,\underline{U+2})}}{\vdots\ \text{evaluate } \mathrm{Add}_2 \atop P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ (K\,\underline{U+2}) \wp (P\,\underline{U+2}) \longrightarrow (\mathsf{apply}\ \mathrm{Add}_2\ V\ K) \wp (P\,U)}}{\vdots\ \forall R, \multimap R \atop P, l, \Sigma_{vs} : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \longrightarrow F'}$$

Using δ_1 and δ_2, I construct δ_3 below. δ_3 is the computation of $(\mathsf{app}\ M'\ \mathrm{Add}_2)$ in FORUM! By the observation regarding δ_1, $S_1(l)$ is a multiple of 2. Hence, it is clear that $(\mathsf{if}\ (=\ 0\,(\mathsf{mod}\,(\mathsf{deref}\ l)\,2))\ \Omega\ 1)$ will evaluate to the value of Ω. Using theorem 3.8 and the fact that Ω is a divergent program, I have proved that $M_{4.2}$ does not converge.

$$\frac{\overset{\delta_2}{P, l : \mathsf{CL}_l\,;\ \longrightarrow F'} \quad \overset{\delta_1}{P, l : F'\,;\ S_1^o(K, V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\ M'\ \mathrm{Add}_2)\ C_1) \wp \Gamma_{S_0} \wp (P\,0)}}{P, l : \mathsf{CL}_l\,;\ S_1^o(K, V') \longrightarrow (\mathsf{eval}\,(\mathsf{app}\ M'\ \mathrm{Add}_2)\ C_1) \wp \Gamma_{S_0} \wp (P\,0)}$$

To complete the proof I have to convert any evaluation of $(\mathsf{app}\ M'\ \mathrm{Add}_2)$ into an evaluation of $(\mathsf{app}\ M'\ c)$. I prove this by a straightforward induction on the height of the evaluation of $(\mathsf{app}\ M'\ \mathrm{Add}_2)$. ■

Example 4.24 [Example 5] Let $M \in \Lambda_{vs}$.

- $M_{5.1} =_{def} (M\,\lambda d.\,\bullet)$
- $M_{5.2} =_{def}$ let val x = (ref 0) in
  let val f = in $\lambda d.$ asg x (+ 1 (deref x))
  $(M\ f)$

■

Lemma 4.25 $M_{5.1} \cong_f M_{5.2}$

Proof: The strategy is to show that an evaluation of $M_{5.1}$ can be transformed to an evaluation of $M_{5.2}$, and vice versa. The essential point of this example is that incrementing achieved by f is useless for M because it can never read the contents of x. Hence, might as well as use $\lambda d.\,\bullet$ instead. This equivalence has been especially problematic for various denotational semantics [OT92].

The proof strategy is the same as in example 4.
Let $F =_{def} \forall K, V.\, (\forall U.\, [(\mathsf{apply}\ c\ V\ K) \wp (P\,U)] \multimap \forall W.\, [(K\,\bullet) \wp (P\,W)])$.
P and c are declared in the signature. F encapsulates the computational behavior of f and $\lambda d.\,\bullet$. Note how it does away with the name of the cell $(P\,U)$, and need for getC and setC clauses for P.

Suppose I have a proof, δ, in FORUM of

$$\Sigma_{vs}, \Sigma_{S_0}, K, c, P, l : \mathcal{E}_{vs}, \mathsf{CL}_{S_0}, F\,;\, S_1^{\circ}(K, V') \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ M'\ c)\ C_1) \wp \Gamma_{S_0} \wp (P\,0),$$

where $E \in EvCont_{vs}$.

- S_0 is a state such that $\mathsf{close}(S_0, E[M])$.
- $V' =_{def} \phi_{vs}(V)$.

- $M' =_{def} \mathcal{H}_{vs}(M)$.
- $C_1 =_{def} \mathcal{C}_{vs}(E, K)$.

I construct δ_i from δ using $CutS$, $i \in [1,2]$. Let $F_i =_{def} F[c := D_i]$, where $D_1 =_{def} \mathcal{H}_{vs}(\lambda d.\, \mathsf{asg}\ l\ (+\ 1\ (\mathsf{deref}\ l)))$, $D_2 =_{def} \mathcal{H}_{vs}(\lambda d.\, \bullet)$, and $i \in [1,2]$.

$$\frac{D_i \text{ is a } \Sigma_{vs}, l \text{ term} \quad \overset{\delta}{c, P, l : F\,;\ S_1^{\circ}(K, V') \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ M'\ c)\ C_1) \wp\, \Gamma_{S_0} \wp\, (P\, 0)}}{P, l : F_i\,;\ S_1^{\circ}(K, V') \longrightarrow (\mathsf{eval}\ (\mathsf{app}\ M'\ D_i)\ C_1) \wp\, \Gamma_{S_0} \wp\, (P\, 0)}$$

I construct δ_3 below. Let $\mathsf{CL}_l =_{def} \{\mathsf{getC}(P, l), \mathsf{setC}(P, l)\}$.

$$\cfrac{\cfrac{\cfrac{\overline{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ (K\, \bullet) \wp\, (P\, \underline{U+1}) \longrightarrow (K\, \bullet) \wp\, (P\, \underline{U+1})}}{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \forall W.\, (K\, \bullet) \wp\, (P\, W) \longrightarrow (K\, \bullet) \wp\, (P\, \underline{U+1})}}{\vdots \text{ evaluate } D_1} }{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \forall W.\, (K\, \bullet) \wp\, (P\, W) \longrightarrow (\mathsf{apply}\ D_1\ V\ K) \wp\, (P\, U)}$$

$$\cfrac{\vdots\ \forall R,\ \multimap\ R}{P, l, \Sigma_{vs} : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \longrightarrow F'}$$

I construct δ_4 below. Let $\mathsf{CL}_l =_{def} \{\mathsf{getC}(P, l), \mathsf{setC}(P, l)\}$.

$$\cfrac{\cfrac{\cfrac{\overline{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ (K\, \bullet) \wp\, (P\, U) \longrightarrow (K\, \bullet) \wp\, (P\, U)}}{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \forall W.\, (K\, \bullet) \wp\, (P\, W) \longrightarrow (K\, \bullet) \wp\, (P\, U)}}{\vdots \text{ evaluate } D_2} }{P, l, K, U, V : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \forall W.\, (K\, \bullet) \wp\, (P\, W) \longrightarrow (\mathsf{apply}\ D_2\ V\ K) \wp\, (P\, U)}$$

$$\cfrac{\vdots\ \forall R,\ \multimap\ R}{P, l, \Sigma_{vs} : \mathsf{CL}_l, \mathcal{E}_{vs}\,;\ \longrightarrow F'}$$

Composing δ_1 and δ_3, I construct the evaluation of $(\mathsf{app}\ M'\ D_1)$ in FORUM. Composing δ_2 and δ_4, I construct δ_6. δ_6 fails to be the evaluation of $(\mathsf{app}\ M'\ D_2)$ in FORUM, because of the extra cell P in the environment. However, I can use lemma 4.16 to get rid of this extra cell. To complete the proof I have to convert proofs of evaluation of $(\mathsf{app}\ M'\ D_i)$, $i \in [1,2]$ to an evaluation of $(\mathsf{app}\ M'\ c)$. I prove this by a straightforward induction on the height of the evaluation of $(\mathsf{app}\ M'\ D_i)$. ■

Using the meta-theory of FORUM, I have proved many of the Meyer-Sieber examples from [MS88] and an example from [OT92]. This style of reasoning bears a close resemblance to the style in [OT93] using logical relations. The use was most remarkable in the fourth and fifth example where I was able to use *CutS* and *CutL* to get the proofs. One direction in this proof still needed to induct on the height of evaluation proof tree, a weakness of the argument that I would like to get rid of in my future work. The first three proofs were essentially arguments about permutations of evaluations in FORUM. I want to investigate whether I can make these proofs compositional using resolution on the proof rules. The meta-theory gives me distinct advantage in the above proofs. However, I would like to develop the meta-theory so that composition, and permutation of resolution can be studied in finer detail.

Chapter 5

Specifying DLX - a RISC architecture

In this chapter I specify the sequential and pipelined operational semantics for the DLX [HP90] architecture – a prototypical RISC (Reduced Instruction Set Computer) architecture – in FORUM. DLX is a generic load/store machine representative of the RISC machines which have become very popular since the late 1980's, *e.g.* Intel i860, MIPS R2000/R3000, Motorola 88000, SPARC, PowerPC. I will prove that the sequential and pipelined specifications of DLX are identical, and using this equivalence give a simple proof of the correctness of code rescheduling. *The main point of this chapter is that FORUM facilitates the declarative specification of the concurrent pipelined operational semantics of* DLX *with complex synchronizations.* Furthermore, the framework allows me to handle *structural* and *data-hazards*, and specify optimizations such as *call-forwarding* and *early-branch prediction* declaratively.

The key feature of the FORUM specification is that it specifies the computation of the pipeline as compared to the existing specifications in the literature which specify the pipeline's temporal behavior [AL93]. The specifications in FORUM are executable as logic programs yielding a prototype implementation of the pipeline which can be used for collecting statistics and experimentation. This seems to be a unique feature of the FORUM specification amongst all the specifications for DLX style pipelines. Furthermore, the FORUM specification is concurrent – different stages of the pipelines can be computed independently of each other. Moreover, the equivalence of sequential and pipeline operational semantics provides me with a tool to prove correct various optimizations done by the back-end optimizer and/or hardware such as code rescheduling. The proofs of program-equivalence are once again achieved by proof transformations very much along the lines of chapter 4.

In this chapter I first introduce the DLX architecture, and specify its sequential operational semantics. I specify only the integer part of DLX and as such, the discussion will be restricted to relevant parts of the architecture, the reader is referred to [HP90] for a detailed description. The specification of the floating-point operations and interrupts in the pipeline do not require any new specification techniques, and thus they have been left out from the present discussion. Next, I specify the operational semantics of DLX pipeline. I prove that the DLX pipeline specification is equivalent to the sequential specification. The program is then extended to call-forwarding and early-branch prediction. Finally, I use the sequential operational semantics to prove the correctness of code scheduling.

5.1 The DLX architecture

The architecture of the DLX machine – the user visible part of the instruction set of DLX – emphasizes design for pipeline efficiency, an easily decoded instruction set and efficiency as a compiler target. In this section, I describe the architecture for the integer part of DLX, for a complete description and discussion of the entire DLX architecture the reader is referred to [HP90]. I have left out the jump instructions because their introduction needs no new ideas – the presence of branch instructions causes all the complications that they may cause.

DLX has thirty-two 32 bit general-purpose registers (GPRs). Memory is word addressable

Instruction	Instruction name	Meaning
lw R1, I(R2)	Load word	R1 := M[I + R2]
add R1, R2, R6	Add	R1 := R2 + R6
sll R1, R2, R4	Shift left logical	R1 := R2 $\ll$ R4
seq R1, R2, R3	Set equal to	if (R2 = R3) $R1 := 1$
		else R1 := 0
beqz R4, $\sharp I$	Branch equal zero	if (R4 = 0) PC := PC + I + 4;
		$((\mathsf{PC}+4)-2^{15}) \leq \mathsf{PC}+I \leq ((\mathsf{PC}+4)+2^{15})$

Figure 5.1: Semantics of example instructions in DLX.

in Big Endian mode with 32-bit address, and all memory references are through loads or stores between memory and the GPRs. I treat memory as word addressable to avoid unilluminating details regarding byte and halfword addressability. All instructions are 32-bits and all memory accesses must be aligned. Since I am only specifying the integer part of DLX, I use the GPRs for integer multiply and divide instructions.

There are three classes of instructions for the integer part of DLX : loads and stores, ALU operations, and branches. A load instruction is written as lw R1, I(R2) with the intended semantics being that R1 is assigned the contents of the memory array, M, from the address R2 plus the 16-bit integer I. A store operation written as sw I(R1), R2 results in M[I + R1] := R2. The operands and results for all ALU operations are stored in registers. The operations include simple arithmetic and logical operations : add, subtract, and, or, exclusive or, shifts and compares. One of the arguments in the operations can be the number itself (called the immediate) instead of a register. However, to focus on the central issues in the specification, I do not consider these variants of the instructions. Typical ALU instruction is written as op R1, R2, R3 with the intended semantics being that R1 is assigned the value R2 op R3.

The branch instructions can only test for equality with zero and the offsets are limited to 16-bit integers. To present the specification in a more understandable way, I have chosen a representative set of instructions from each class for the DLX [HP90]. I have added instructions end and begin, which cause the computation to halt and start, respectively.

<u>Data transfers</u>	<u>Move data between registers and memory</u>
lw, sw	Load-word, store word (to/from GPR).
<u>Arithmetic/Logical</u>	<u>Operations on integer or logical data in GPRs</u>
add, addu, sub, subu	Add and subtract; signed and unsigned.
mult, div, multu, divu	Multiply and divide; signed and unsigned.
and, or, xor	And, or, exclusive or.
sll, srl, sra	Shifts: left and right logical, and right arithmetic.
s__	Set conditional: "__" may be lt, le, eq, ne.
<u>Control</u>	<u>Conditional branches</u>
beqz, bnez	Branch GPR equal/not equal to zero; 16-bit offset from $\mathsf{PC}+1$.
end, begin	Halt, start computation.

Figure 5.2: List of DLX instructions selected for specification.

A sample of instructions from the different classes along with their intended semantics is given in figure 5.1. A list of the selected DLX instructions is given in figure 5.2.

5.2 Sequential specification for DLX architecture

In this section, I specify the sequential semantics of the DLX machine. The DLX instructions are of a very simple nature. In particular, no instruction can both perform an arithmetic operation and a memory operation. Consequently all instructions can be broken into five distinct parts: fetch the instruction to be executed, decode instruction, execute instruction, perform required memory-access, and write-result. The block diagram detailing the connectivity of the various units is shown in figure 5.3. In the figure only MAR (Memory Address Register) can set the address for a memory load/store, and only MDR (Memory Data Register), can send/receive data from memory. IR (Instruction Register), and PC (Program Counter), save the current instruction being executed and the address of the next instruction to be executed, respectively. The latches A, B and AOUT provide storage for the inputs and the outputs of the ALU. The block named CONTROL decodes the instruction in IR, and sets all the switches to generate the appropriate flow of data required

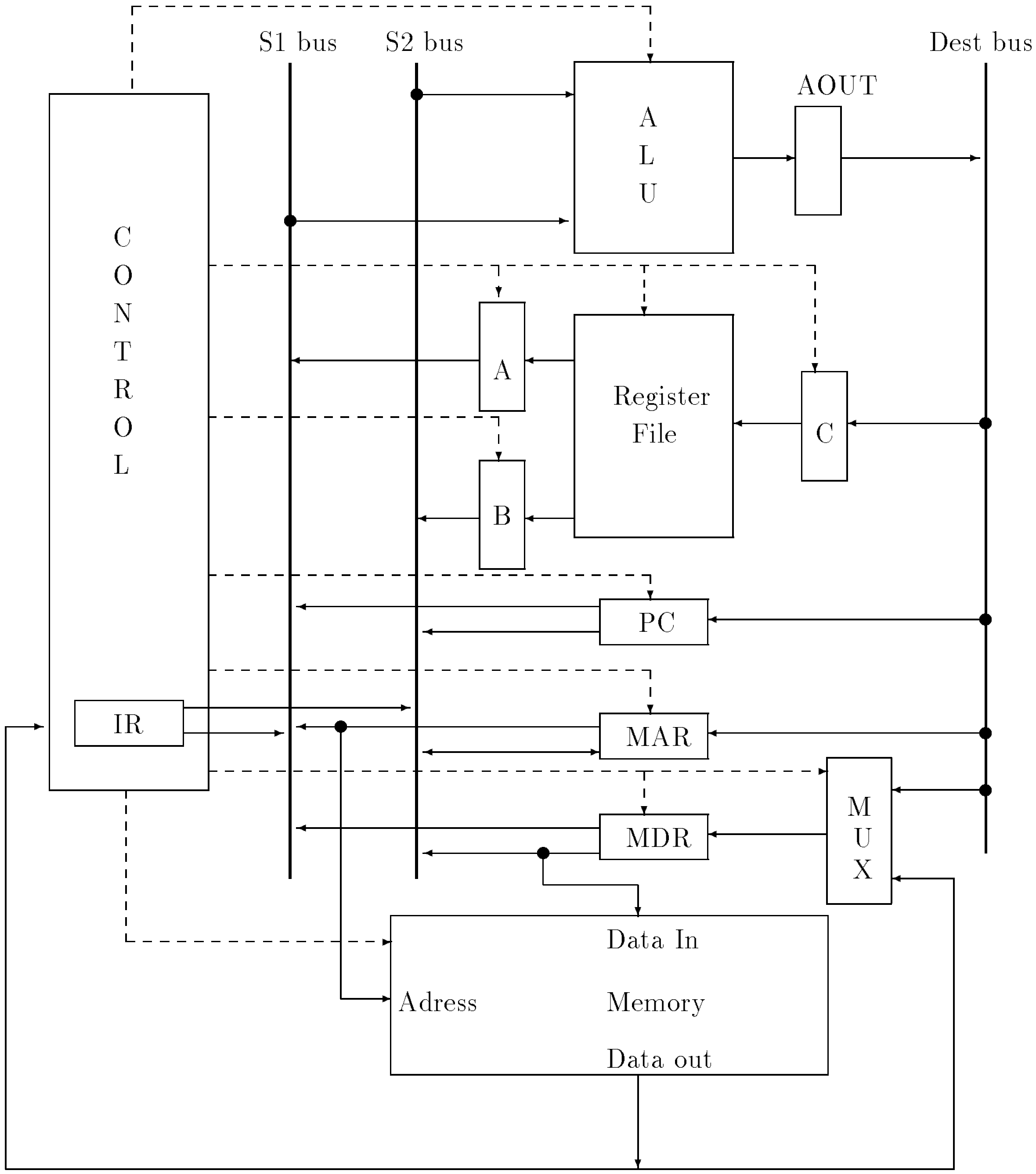

Figure 5.3: Block diagram for the connectivity of functional blocks in the DLX.

to execute the current instruction. The individual parts in the execution of an instruction are further elaborated below.

1. Instruction fetch (IF): $\mathsf{MAR} := \mathsf{PC}$; $\mathsf{IR} := \mathsf{M[MAR]}$

   Operation: Send out the PC and fetch the instruction from memory into the instruction register, IR. PC is transferred to MAR because PC is connected to memory only via MAR.

2. Instruction decode/register fetch (ID): $\mathsf{A} := \mathsf{Rs1}$; $\mathsf{B} := \mathsf{Rs2}$; $\mathsf{PC} := \mathsf{PC} + 1$

   Operation: Decode the instruction and access the source registers from the register file. The PC is also incremented to point to the next instruction to be executed. Decoding is done in parallel with reading registers to the latches A and B, because of the fixed format of the DLX instructions. Moreover, as the immediate argument occurs in the same bits in all DLX instructions, the sign-extended immediate, if needed, is also calculated in this step.

3. Execution/effective address (EX): The ALU operates on the operands, performing one of the following functions depending upon the DLX instruction type.

   - Memory reference: $\mathsf{MAR} := \mathsf{A} + (\mathsf{IR}_{16})^{16}\sharp\sharp\mathsf{IR}_{16\ldots31}$; $\mathsf{MDR} := \mathsf{Rd}$

     Operation: The immediate is calculated by taking the upper 16-bits of the IR and filling the lower 16-bits of the immediate with the 16th bit of the IR. The immediate is added to the latch A. The destination register Rd is stored in MDR, because GPRs can store data into memory only via MDR. $\mathsf{IR}_{16\ldots31}$ are the lower 16 bits of IR, and $(\mathsf{IR}_{16})^{16}$ is the 16th bit of IR repeated 16 times.

   - ALU instruction: $\mathsf{AOUT} := \mathsf{A}\ \mathsf{op}\ \mathsf{B}$

     Operation: The ALU performs the operation specified in the opcode on the value in latch A and on the value in B. The result is stored in another latch called AOUT.

   - Branch: $\mathsf{AOUT} := \mathsf{PC} + (\mathsf{IR}_{16})^{16}\sharp\sharp\mathsf{IR}_{16\ldots31}$; cond $:= (\mathsf{A}\ \mathsf{op}\ 0)$

     Operation: The ALU computes the branch target address by adding the PC to the immediate. It then compares A to 0; op can be either $=$ or $\neq$.

4. Memory access/branch completion (MEM): The only DLX instructions active in this step are loads, stores and branches.

   - Memory reference: MDR := M[MAR] or M[MAR] := MDR

     Operation: For a load data comes from memory to MDR, and for a store data from MDR goes to memory.

   - Branch: if (cond) PC := AOUT

     Operation: For branch instructions PC is updated if cond is 1.

5. Write-back (WB): Rd := AOUT *or* MDR

   Operation: Write the result into the register file, whether coming from the memory system or from the ALU.

The idea behind specifying DLX is pretty clear, given the above explanation of the operation semantics of the instructions. I look at the various registers and the memory array as entities in some common pool. The evaluation of the program consists of a synchronization between the PC, the registers, and the memory, which contains both the program and the data. Memory is represented by a two place predicate, ($\mathtt{m}\, n\, V$), the first argument is the address - an integer - of the memory cell and the second argument is the contents of the cell. Similarly, registers are represented by binary predicates ($\mathtt{r}\, i\, V$) and program memory by ($\mathtt{p}\, i\, x$). An instruction is represented as ($\mathtt{ix}\;\; X\, s_1\, s_2\, d\, I\, \mathsf{op}$), where X denotes the class of the instruction, s_1 the first register argument, s_2 the second register argument, d the destination register, I is an immediate and op the particular function to calculate. It should be obvious that all instructions can be represented with this representation, however, it is not necessary that all fields will have meaningful data for all the instructions.

The signature, called Σ_s, specifying the constants used in the sequential specification of DLX is given in figure 5.4. A function for every arithmetic/logical function of DLX is assumed in FORUM – I treat these functions as if they were in-built. Further, the type `int` is declared with all the integers as terms of type `int`. The built-in functions are represented by tokens which are members of `func`. Members of the type `class` represent the classes of instructions. The predicate `cont` signals that computation should continue, and `num` is a predicate whose argument counts the number of instructions executed. $(\mathtt{eq}\, N)$ is provable if and only N is 1, and $(\mathtt{ne}\, N)$ is provable if and only if N is 0.

$$\begin{array}{rcl}
\mathtt{cont} & : & o \\
\mathtt{num}, \mathtt{eq}, \mathtt{ne} & : & \mathtt{int} \rightarrow o \\
\mathtt{op} & : & \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \\
\mathtt{r}, \mathtt{m} & : & \mathtt{int} \rightarrow \mathtt{int} \rightarrow o \\
\mathtt{a}, \mathtt{b}, \mathtt{pc} & : & \mathtt{int} \rightarrow o \\
\mathtt{p} & : & \mathtt{int} \rightarrow \mathtt{inst} \rightarrow o \\
\mathtt{ix} & : & \mathtt{class} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{func} \rightarrow \mathtt{inst} \\
\underline{\mathtt{op}} & : & \mathtt{func} \\
\mathtt{alu}, \mathtt{br}, \mathtt{ld}, \mathtt{st}, \mathtt{ht}, \mathtt{bg}, \mathtt{noop} & : & \mathtt{class}
\end{array}$$

Figure 5.4: Signature for specification of DLX

The specification for DLX, called $\mathcal{E}_s$, is the set of universal closure of clauses in figure 5.6. Different kinds of parentheses have been used in figure 5.6 to enhance readability. Note that for every instruction type – specified by its class – there is exactly one clause in figure 5.6 with a matching head. The one subtlety in the specification is that I cannot perform a multi-way synchronization amongst the operand registers, destination register, and PC for the ALU operations. The problem is that there is only one Rn, while the instruction may need three copies – a deadlock would occur. The specification thus decouples the reading of the two source registers into individual unsynchronized steps. The idea is that the two source registers are read in any order into the latches A and B. Once the latches are loaded the operation is performed and result is stored in the destination register. In this approach all of the register arguments may be identical or different – a deadlock will not occur.

The DLX programs in FORUM are defined by the terms parsed by the non-terminal P_l, $l \in \mathsf{nat}$ in figure 5.5. A program is loaded in program memory, and then evaluated with respect to a state specified by the contents of the special purpose registers, general purpose registers, memory and PC – the data state. The definitions of program state and data state are given below.

Definition 5.1 [Program State] Let $\mathcal{P}$ be an abbreviation for
$\lambda m, x_1, \ldots, x_m. (\mathsf{p}\, 1\, x_1) \wp \ldots \wp (\mathsf{p}\, m\, x_m)$ for $m \in \mathsf{nat}$. *For any dlx program* P_l, $(\mathcal{P}\ l\ \vec{P})$ *is a*

$$
\begin{array}{rcll}
R & ::= & n & n \in [1,32] \\
C & ::= & \texttt{ld} \mid \texttt{st} \mid \texttt{alu} & \\
I_{l',l,m,n} & ::= & q & q \in [l',l] \cup [0,m] \cup [-n,0] \\
Ix_{l',l,m,n} & ::= & (\texttt{ix br}\ R\ R\ R\ I_{l',l,m,n}\ \underline{\texttt{op}}) & \\
 & & \mid (\texttt{ix}\ C\ R\ R\ R\ I_{0,0,q,q}\ \underline{\texttt{op}}) & q = 2^{15} \\
B_{1,l,m,n} & ::= & Ix_{m,l-n-1,0,n+2} & \\
B_{l',l,m,n} & ::= & Ix_{m+l'-1,l-n-1,l'-1,n+2}; B_{l'-1,l,m,n+1} & (l'+n), m < l \text{ and } 1 < l' \\
A_{1,l,m,n} & ::= & Ix_{-(l+1),-(m+n+2),0,n+1} & \\
A_{l',l,m,n} & ::= & Ix_{-(l+2-l'),-(m+n+2),l'-1,-(n+1)}; A_{l'-1,l,m,n+1} & (l'+n), m < l \text{ and } 1 < l' \\
H_{1,l} & ::= & Ix_{0,0,-1,l} & \\
H_{l',l} & ::= & Ix_{0,0,l'-2,l-l'-1}; H_{l'-1,l} & 1 < l' \leq l \\
S & ::= & (\texttt{ix bg}\ R\ R\ R\ I_{0,0}\ \underline{\texttt{op}}) & \\
E & ::= & (\texttt{ix ht}\ R\ R\ R\ I_{0,0}\ \underline{\texttt{op}}) & \\
Q_2 & ::= & S : E & \\
Q_{l+2} & ::= & S; B_{l,l,0,0}; E & 1 \leq l \\
Q_{l+2} & ::= & S; B_{l',l,m,0}; H_m; A_{l'',l,m,0}; E & l = l' + l'' + m, 1 \leq l', l'', m \\
P_l & ::= & Q_l \mid Q_{l'}; P_{l''} & l = l' + l'', \text{ and } 2 \leq l', l''
\end{array}
$$

Figure 5.5: Grammar for DLX programs

program state. $\vec{P}$ are the instructions in P_l, and by definition l is the number of instructions in $\vec{P}$. ■

Definition 5.2 [Data State] $(\texttt{r}\,1\,)\ldots(\texttt{r}\,32\,)$ are the DLX registers, $n \in \textsf{nat}$ is the number of memory cells, `pc` is the program counter, and `num` stores the number of instructions executed. Let $\mathcal{S}$ be an abbreviation for

$$\lambda n, \vec{V}, \vec{U}, L.\, (\texttt{pc}\, L) \wp (\texttt{r}\, 1\, V_1) \wp \ldots (\texttt{r}\, 32\, V_{32}) \wp (\texttt{m}\, 1\, U_1) \wp \ldots (\texttt{m}\, n\, U_n) \wp (\texttt{num}\, U_{n+1}).$$

The lengths of $\vec{V}$ and $\vec{U}$ – 32 and $n+1$ respectively – if implicit, are assumed to be as required.

For any $n \geq 0$, $L, V_1, \ldots, V_{32}, U_1, \ldots, U_{n+1}$: `int`, $(\mathcal{S}\ n\, \vec{V}\, \vec{U}\, L)$ *is a data state.* ■

It is worth pointing out that execution cannot start without executing `bg` as clauses for all instructions other than `bg` synchronize with `cont`, and that the execution cannot terminate

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ alu}\,S_1\,S_2\,D\,I\,O))\wp\,\mathtt{cont}\,\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad[(\mathtt{p}\,L\,(\mathtt{ix\ alu}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{num}(M+1))\ \circ\!\!-\\
\qquad\quad[[(\mathtt{r}\,S_1\,V_1)\ \circ\!\!-\ (\mathtt{r}\,S_1\,V_1)\wp(\mathtt{a}\,V_1)]\\
\qquad\quad\ \otimes[(\mathtt{r}\,S_2\,V_2)\ \circ\!\!-\ (\mathtt{r}\,S_2\,V_2)\wp(\mathtt{b}\,V_2)]\\
\qquad\quad\ \otimes[(\mathtt{r}\,D\,V_3)\wp(\mathtt{a}\,V_1)\wp(\mathtt{b}\,V_2)\ \circ\!\!-\ (\mathtt{r}\,D\,(V_1\ O\ V_2))\wp\,\mathtt{cont}\,\wp(\mathtt{pc}\,(L+1))]]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ ld}\,S_1\,S_2\,D\,I\,O))\wp\,\mathtt{cont}\,\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad[(\mathtt{p}\,L\,(\mathtt{ix\ ld}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{num}(M+1))\ \circ\!\!-\\
\qquad\quad[[(\mathtt{r}\,S_1\,V_1)\ \circ\!\!-\ (\mathtt{r}\,S_1\,V_1)\wp(\mathtt{a}\,V_1)]\\
\qquad\quad\ \otimes[(\mathtt{r}\,D\,V_2)\wp(\mathtt{a}\,V_1)\wp(\mathtt{m}\,(V_1+I)\,V_3)\ \circ\!\!-\\
\qquad\qquad\quad(\mathtt{r}\,D\,V_3)\wp\,\mathtt{cont}\,\wp(\mathtt{pc}\,(L+1))\wp(\mathtt{m}\,(V_1+I)\,V_3)]]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ st}\,S_1\,S_2\,D\,I\,O))\wp\,\mathtt{cont}\,\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad[(\mathtt{p}\,L\,(\mathtt{ix\ st}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{num}(M+1))\ \circ\!\!-\\
\qquad\quad[[(\mathtt{r}\,S_1\,V_1)\ \circ\!\!-\ (\mathtt{r}\,S_1\,V_1)\wp(\mathtt{a}\,V_1)]\\
\qquad\quad\ \otimes[(\mathtt{r}\,D\,V_2)\wp(\mathtt{a}\,V_1)\wp(\mathtt{m}\,(V_1+I)\,V_3)\ \circ\!\!-\\
\qquad\qquad\quad(\mathtt{r}\,D\,V_2)\wp\,\mathtt{cont}\,\wp(\mathtt{pc}\,(L+1))\wp(\mathtt{m}\,(V_1+I)\,V_2)]]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ br}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{r}\,S_1\,V_1)\wp\,\mathtt{cont}\,\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad(\mathtt{p}\,L\,(\mathtt{ix\ br}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{r}\,S_1\,V_1)\wp\,\mathtt{cont}\,\wp(\mathtt{num}(M+1))\wp\\
\quad[[(\mathtt{eq}\,(V_1\ O\ 0))\otimes(\mathtt{pc}\,(L+1+I))]\oplus[(\mathtt{ne}\,(V_1\ O\ 0))\otimes(\mathtt{pc}\,(L+1))]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ ht}\,S_1\,S_2\,D\,I\,O))\wp\,\mathtt{cont}\,\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad(\mathtt{pc}\,(L+1))\wp(\mathtt{p}\,L\,(\mathtt{ix\ ht}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{num}(M+1))
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,(\mathtt{ix\ bg}\,S_1\,S_2\,D\,I\,O))\wp(\mathtt{num}\,M)\ \circ\!\!-\\
\quad(\mathtt{pc}\,(L+1))\wp(\mathtt{p}\,L\,(\mathtt{ix\ bg}\,S_1\,S_2\,D\,I\,O))\wp\,\mathtt{cont}\,\wp(\mathtt{num}(M+1))
\end{array}
$$

$$(\mathtt{eq}\,1)\ \circ\!\!-\ \mathbf{1}$$

$$(\mathtt{ne}\,0)\ \circ\!\!-\ \mathbf{1}$$

Figure 5.6: Sequential specification of DLX

successfully without executing `ht` because the clause for `ht` is the only one that consumes `cont`. Hence, the value of `pc` in the initial data state must address a `bg` instruction in the program state. The program state is static – its contents do not change during execution – it instructs the machine how to alter the data state. Thus, the evaluation, given a program state, transforms one data state into another. The definition of evaluation is made precise below.

Definition 5.3 [Sequential evaluation in DLX, $\mathcal{E}_s$] Given data states $\mathcal{S}_1$ and $\mathcal{S}_2$, and $(\mathcal{P}\ l\ \vec{P}_l)$, a program state. P_l evaluates in $\mathcal{S}_1$ to $\mathcal{S}_2$ written as $\mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_s \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l)$, if

$$\Sigma_s : \mathcal{E}_s\, ; \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l) \longrightarrow \mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l)$$

is provable in FORUM. ■

A small example of an evaluation will explain the specification better. I consider a program that adds the contents of second and third register, and places the result in the second register. The program is stored in the memory starting at the first cell. The data state, and the program state are described in figure 5.7. I use $\mathcal{S}_1$ as an abbreviation for the entire expression in the figure. At the end of the computation the resultant state, $\mathcal{S}_2$ will have 9 in the second register, 4 in the `pc`, 3 in the `num` and otherwise be identical to $\mathcal{S}_1$. The proof of the computation is detailed below.

$$\begin{array}{l}(\mathtt{pc}\ 0) \wp (\mathtt{r}\ 1\ 0) \wp (\mathtt{r}\ 2\ 4) \wp (\mathtt{r}\ 3\ 5) \wp \ldots \wp (\mathtt{r}\ 32\ V_{32}) \wp (\mathtt{m}\ n\ \vec{U_n}) \wp (\mathtt{num}\ 0) \wp \\ (\mathtt{p}\ 1\ (\mathtt{ix\ bg}\ 1\ 2\ 3\ 0\ +)) \wp (\mathtt{p}\ 2\ (\mathtt{ix\ alu}\ 2\ 3\ 2\ I\ +)) \wp (\mathtt{p}\ 3\ (\mathtt{ix\ ht}\ 1\ 2\ 3\ 0\ +)) \wp (\mathtt{p}\ m\ \vec{W_m})\end{array}$$

Figure 5.7: Example program in DLX

Let $\mathcal{S}_1'$ be identical to $\mathcal{S}_1$, except that it does not contain `pc` and `num`, and $\mathcal{S}_1''$ be identical to $\mathcal{S}_1$ except that it does not contain `pc`, $(\mathtt{r}\ 2)$ and `num`. I begin the proof by backchaining on `bg` clause, which generates `cont`. Next, I backchain on the `alu` clause, and then the only clauses that I can use for backchaining are the ones introduced by the `alu` clause to read the two source operands. Hence, I read second and third registers into `a` and `b`, respectively. Next the actual computation is performed using the values in the latches, and the value of $(\mathtt{r}\,2\,)$ is updated to 9. To complete the proof, I have to construct δ. $\mathcal{E}_s$ and Σ_s are not written in the proofs for the sake of readability. It is also worth pointing out that neither provability nor the actual answers computed depend upon the order in which the source registers are read.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{\delta}{\mathcal{S}_2 \longrightarrow \mathcal{S}_1'' \wp (\mathtt{r}\,2\,9) \wp\, \mathtt{cont} \wp (\mathtt{pc}\,3) \wp (\mathtt{num}\,2)}}{\mathcal{S}_2, (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \wp (\mathtt{b}\,5) \mathrel{\circ\!\!-} (\mathtt{r}\,2\,9) \wp\, \mathtt{cont} \wp (\mathtt{pc}\,3) \longrightarrow \mathcal{S}_1' \wp (\mathtt{b}\,5) \wp (\mathtt{a}\,4) \wp (\mathtt{num}\,2)}}{\begin{array}{l} (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \wp (\mathtt{b}\,5) \mathrel{\circ\!\!-} (\mathtt{r}\,2\,9) \wp\, \mathtt{cont} \wp (\mathtt{pc}\,3), \\ \qquad \mathcal{S}_2, (\mathtt{r}\,2\,4) \mathrel{\circ\!\!-} (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \qquad \longrightarrow \mathcal{S}_1' \wp (\mathtt{b}\,5) \wp (\mathtt{num}\,2) \end{array}}}{\begin{array}{l} (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \wp (\mathtt{b}\,5) \mathrel{\circ\!\!-} (\mathtt{r}\,2\,9) \wp\, \mathtt{cont} \wp (\mathtt{pc}\,3), \\ \qquad (\mathtt{r}\,3\,5) \mathrel{\circ\!\!-} (\mathtt{r}\,3\,5) \wp (\mathtt{b}\,5), \\ \qquad \mathcal{S}_2, (\mathtt{r}\,2\,4) \mathrel{\circ\!\!-} (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \qquad \longrightarrow \mathcal{S}_1' \wp (\mathtt{num}\,2) \end{array}}}{\vdots\, \mathtt{alu}\ \textit{clause},\ \multimap\text{-}R,\ \otimes\text{-}L}}{\dfrac{\mathcal{S}_2 \longrightarrow (\mathtt{pc}\,2) \wp \mathcal{S}_1' \wp\, \mathtt{cont} \wp (\mathtt{num}\,1)}{\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_2 \longrightarrow \mathcal{S}_1}\ \mathtt{bg}\ \textit{clause}}
$$

To construct δ notice that by definition of $\mathcal{S}_2$, the state $\mathcal{S}_1'' \wp (\mathtt{r}\,2\,9) \wp (\mathtt{pc}\,4) \wp (\mathtt{num}\,3)$ is identical to $\mathcal{S}_2$ upto associativity and commutativity of $\wp$ – the states are being treated as multisets. Hence, δ is constructed by first backchaining on the `ht` clause, and then using the identity rule. The proof is detailed below. $\mathcal{E}_s$ and Σ_s are not written in the proofs for sake of readability.

$$
\dfrac{\dfrac{}{\mathcal{S}_2 \longrightarrow \mathcal{S}_1'' \wp (\mathtt{r}\,2\,9) \wp (\mathtt{pc}\,4) \wp (\mathtt{num}\,3)}\ \textit{Initial}}{\mathcal{S}_2 \longrightarrow \mathcal{S}_1'' \wp (\mathtt{r}\,2\,9) \wp\, \mathtt{cont} \wp (\mathtt{pc}\,3) \wp (\mathtt{num}\,2)}\ \mathtt{ht}\ \textit{clause}
$$

One consequence of definition 5.3 is that evaluation can be composed using cuts. Suppose $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ are three states, and $\mathcal{P}$ is a program state such that $\mathcal{S}_1 \wp \mathcal{P} \mapsto_s \mathcal{S}_2 \wp \mathcal{P}$ and $\mathcal{S}_2 \wp \mathcal{P} \mapsto_s \mathcal{S}_3 \wp \mathcal{P}$. The computation for $\mathcal{S}_1 \wp \mathcal{P} \mapsto_s \mathcal{S}_3 \wp \mathcal{P}$ is obtained by a linear cut on δ_1 and δ_2, which are provided by definition 5.3.

$$
\dfrac{\overset{\delta_2}{\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_3 \wp \mathcal{P} \longrightarrow \mathcal{S}_2 \wp \mathcal{P}} \quad \overset{\delta_1}{\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_2 \wp \mathcal{P} \longrightarrow \mathcal{S}_1 \wp \mathcal{P}}}{\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_3 \wp \mathcal{P} \longrightarrow \mathcal{S}_1 \wp \mathcal{P}}\ \textit{CutL}
$$

Hence, proofs of evaluations for the DLX machine will be composed using cuts – I use the cut-elimination theorem for FORUM. *The main point is that the computation of an entirely imperative program is being represented declaratively and analyzed by proof theoretic tools such as cut-elimination.*

5.3 Pipelining DLX - facing the hazards

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is a key technique used to make fast CPU's (Central Processing Unit). The basic idea of pipelining was first implemented in IBM 7030 [Blo59, Buc62]. The CDC 6600 [Tho70] and IBM 360/91 [AST89] introduced many important concepts in pipelining, including scoreboarding, use of multiple functional units, simple architecture for efficient pipelining, and tagging of data, dynamic memory hazard resolution, and generalized forwarding. With the advent of RISC [AC87], efficient pipelining and compilation became integral parts of the architecture design. Many new ideas and designs for RISC architectures have been explored in the last decade, resulting in the design of important machines such as the Intel i860, MIPS R2000/R3000, Motorola 88000, SPARC, PowerPC. As we see in figure 5.8, all instructions in DLX have five distinct parts. The intention is to execute the five distinct parts of five instructions at the same time, and complete the execution of an instruction every time unit. Since the time unit for the pipeline will be determined by the execution time of its slowest segment it is important to have similar work loads for the different stages of the pipeline. Furthermore, since the performance gain for the pipeline will be maximized if all stages of the pipeline are kept busy it is important to minimize stalls in the pipeline.

Pipelining exploits the simple nature of the DLX architecture which facilitates well balanced pipeline segments, and early detection and elimination of possible stalls in the pipeline. Although the intentions are concisely stated, the design of pipelines is a tight rope walk, balancing various parameters to improve performance. The DLX instructions are not seen as atomic entities any longer - the five stages, *i.e.* IF, ID, EX, MEM and WB, are the atomic

	ALU	**LOAD**	**STORE**	**CONTROL**
IF	$\mathsf{IR} := \mathsf{M[PC]}$ $\mathsf{PC} := \mathsf{PC} + 1$	$\mathsf{IR} := \mathsf{M[PC]}$ $\mathsf{PC} := \mathsf{PC} + 1$	$\mathsf{IR} := \mathsf{M[PC]}$ $\mathsf{PC} := \mathsf{PC} + 1$	$\mathsf{IR} := \mathsf{M[PC]}$ $\mathsf{PC} := \mathsf{PC} + 1$
ID	$\mathsf{A} := \mathsf{Rs1}$ $\mathsf{B} := \mathsf{Rs2}$ $\mathsf{PC}_1 := \mathsf{PC}$ $\mathsf{IR}_1 := \mathsf{IR}$	$\mathsf{A} := \mathsf{Rs1}$ $\mathsf{B} := \mathsf{Rs2}$ $\mathsf{PC}_1 := \mathsf{PC}$ $\mathsf{IR}_1 := \mathsf{IR}$	$\mathsf{A} := \mathsf{Rs1}$ $\mathsf{B} := \mathsf{Rs2}$ $\mathsf{PC}_1 := \mathsf{PC}$ $\mathsf{IR}_1 := \mathsf{IR}$	$\mathsf{A} := \mathsf{Rs1}$ $\mathsf{B} := \mathsf{Rs2}$ $\mathsf{PC}_1 := \mathsf{PC}$ $\mathsf{IR}_1 := \mathsf{IR}$
EX	$\mathsf{AOUT} := \mathsf{A}\ \mathsf{op}\ \mathsf{B}$	$\mathsf{MAR} := \mathsf{A} +$ $(\mathsf{IR}_{116})^{16} \sharp\sharp \mathsf{IR}_{116\ldots31}$	$\mathsf{MAR} := \mathsf{A} +$ $(\mathsf{IR}_{116})^{16} \sharp\sharp \mathsf{IR}_{116\ldots31}$ $\mathsf{SMDR} := \mathsf{B}$	$\mathsf{AOUT} := \mathsf{PC}_1 +$ $(\mathsf{IR}_{116})^{16} \sharp\sharp \mathsf{IR}_{116\ldots31}$ $\mathsf{cond} := (\mathsf{Rs1}\ \mathsf{op}\ 0)$
MEM	$\mathsf{AOUT}_1 := \mathsf{AOUT}$	$\mathsf{LMDR} := \mathsf{M[MAR]}$	$\mathsf{M[MAR]} := \mathsf{SMDR}$	$\mathsf{if}\ (\mathsf{cond})\ \{$ $\mathsf{PC} := \mathsf{AOUT}\}$
WB	$\mathsf{Rd} := \mathsf{AOUT}_1$	$\mathsf{Rd} := \mathsf{LMDR}$		

Figure 5.8: DLX pipeline structure

activities. The parallel execution of these atomic steps for different instructions requires complex control to ensure that the results computed are identical to the ones computed by $\mathcal{E}_s$.

I view pipelining as an alternate operational semantics for DLX programs – a different evaluator for DLX programs. The declarative specification of the pipelines which can be executed to simulate pipelined computation has not been attempted to the best of my knowledge. The thrust of the existing work has been either to verify the correctness of a pipelined processor all the way down to the circuit level [TK93], or to automate the production of control circuitry from high level descriptions of pipelines [AL93]. My goal is to provide an executable and declarative specification of pipelines explaining the intricate synchronizations required to implement the basic concepts in modern pipelined architectures.

In this section, I begin by introducing the basic pipeline structure for DLX, and structural and data hazards. The pipeline is then specified using external functions to resolve the hazards. The specification in this section does not incorporate any of the standard techniques to improve the CPU throughput. The implementation of call-forwarding and early branch resolution will be the subject of the next section. Furthermore, I prove that the pipeline specification is equivalent to the sequential specification, $\mathcal{E}_s$.

Hazards in Pipeline

I model IF, ID, EX, MEM and WB and the clock as cyclic processes. The clock generates signals prompting the stages to start their respective computations. When all the stages complete their designated computations they notify the clock, which once again generates the enabling signal for the stages, and the cycle continues. Although there is no synchronization amongst different stages, the computation for every stage is synchronized via the clock signal.

Once the processes for the clock and the pipeline stages start running in parallel, it is possible to run into a variety of problems. Imagine that a machine can only write/read to one register at a time. In such a machine executing the WB stage and ID stage of two `alu` instructions would cause a contention for register port. The point is that when execution of stages is done in parallel, certain resources need to be duplicated to avoid conflicts over resource usage. Existence of such a conflict can stall the pipeline because one instruction will have to wait while the other uses the resource, causing the entire pipeline to waste one time unit. Such conflicts are called structural hazards. *Informally, if there is a combination of instructions which causes contention for resources, then the machine is said to have structural hazards.*

In the DLX machine that I outline there will be no structural hazards, essentially because all instructions are assumed to complete in one time unit and resources have been duplicated sufficiently. Although eliminating structural hazards from pipelines yields better throughput, designers sometimes allow structural hazards since, either duplicating resources is too expensive or eliminating them results in a larger time unit for the pipeline.

The concurrent execution of parts of DLX instructions alters the relative timing of instructions. *In the sequential specification an instruction is executed only after the preceding instruction is completed, whereas, in the pipeline there are up to five instructions whose various parts are being computed at the same time.* So for example (`ix ld` $1\,2\,2\,I$ `op`) could be in the EX stage while (`ix alu` $1\,2\,4\,I+$) is in the ID stage. Now I have a problem, since (`ix alu` $1\,2\,4\,I+$) reads the value in R2 before the `ld` has fetched the contents of memory addressed by R1 and stored them in R2, which happens at the end of the WB stage (figure 5.8). The problem is that there is a data-dependence in the sequence of instructions above. It causes no problems for $\mathcal{E}_s$, because `alu` starts only after the completion of `ld`. Thus the pipeline must be stalled while `alu` waits for `ld` to finish. Such problems are significantly eliminated by code rescheduling and forwarding [HP90].

Definition 5.4 [Data Hazard] If the order of access to operands by instructions is changed due to overlapping execution in pipeline, then there is a data hazard. Data Hazards are classified by the order in which the read and writes are supposed to occur in the program. Let i be an instruction occurring before j in the execution.

- RAW (read after write) : j tries to read a source before i writes it, and thus gets the old value.
- WAR (write after read) : j tries to write a destination before i has read it, and thus i gets the new value.
- WAW (write after write) : j tries to write an operand before it is written by i, and thus the writes are performed in the wrong order.

■

Note that RAR (read after read) is not an error. In the DLX pipeline stages all instructions complete in one time unit, and registers are read early at ID as compared to written later at WB. Moreover the memory accesses are kept in order, and hence no WAR hazard is possible in DLX. Since the WAW hazard is only possible in pipelines that write in more than one stage of the pipeline, this hazard is avoided by DLX which writes registers only in the WB stage. The only kind of data hazard in the DLX pipeline is RAW, as exhibited in the example above regarding the `ld` and `alu` instructions.

The concurrent execution of instructions causes another kind of synchronization problem. Suppose (`ix br` $1\,2\,4\,I\ \ =$) enters the ID stage. The IF stage now needs to fetch the next instruction to be executed, but this is not possible before the branch instruction is resolved and new `pc` is available at the end of the MEM stage, figure 5.8. Moreover, at the beginning of the IF stage it is not known whether the previous instruction was a `br` or not. Thus the IF fetches the next instruction, and invalidates the fetched instruction if the preceding instruction is a `br` instruction. The pipeline is stalled until the end of the MEM stage for the `br` instruction when the new `pc` is available. The fact that I fetch and invalidate an instruction introduces wasteful computation and is different in nature from the stall introduced by data hazards. *Informally, if the pipeline is stalled for a* `br` *instruction to calculate the new* `pc`*, then there is a control hazard.*

Even in the simple setting of the integer DLX architecture hazards and stalls arise due to the concurrent nature of the pipeline. The presence of instructions which do not complete in one time unit and interrupts further complicates the control circuitry. However, the specification techniques used to resolve the hazards for the simple unoptimized pipeline present the ideas needed to tackle the various synchronization and control issues that arise in the specification. To keep the presentation concise and convey the essential ideas, I chose to specify only the integer DLX with call-forwarding and early-branch resolution, and not deal with floating-point arithmetic and interrupts. Floating-point arithmetic and interrupts are challenging aspects of pipeline design, the comments above are only with respect to the specification techniques used to specify the pipeline design for these features.

Specifying the pipeline

Given the hazards in DLX pipeline, it will not be enough to specify the processes representing the pipeline stages and the clock. The crucial ingredient in the specification is the synchronization which avoids all the hazards that may occur during execution. The resource conscious language of FORUM provides an ideal setting to specify the complex interactions required to specify the pipeline.

I begin by giving the signature, Σ_p, for the specification of the pipeline. The signature is the union of Σ_s and the constants in figure 5.9. `lr` stores the data to be written back to the register file. The `cond` predicate stores conditional value for branch instructions, `mar` stores the memory address to be accessed, `lmdr` stores the data read from the memory and `smdr` stores the data to be written into memory. The `aout` predicate stores the ALU result at the end of the EX, and $\mathtt{aout}_1$ stores the ALU result at the end of the MEM. The $\mathtt{pc}_1$ predicate stores the old `pc` to EX, `ir` stores the instruction, and `it` stores the class, destination, immediate and operator of the instruction. The predicates with a `l` before the above predicate names are latches which hold temporary values in between reading and writing to the registers. The beginning of the read and write pahse for IF are signalled by `ifrb` and `ifwb`, respectively, while `ifrd` and `ifwd` signal the completion of the read and write phase, respectively, for IF. Similarly there are predicates for ID, EX, MEM and WB prefixed by their names. The class of an instruction is tested by `alu`?, `ld`?, `st`?, `br`? and `noop`? predicates.

The following definitions make the presentation more concise. I sometime use $\vec{n}$ as an abbreviation for n_1, n_2, n_3, n_4, n_5.

- $\mathtt{crb} =_{def} \lambda\vec{n}.\,(\mathtt{ifrb}\,n_1)\,\wp\,(\mathtt{idrb}\,n_2)\,\wp\,(\mathtt{exrb}\,n_3)\,\wp\,(\mathtt{merb}\,n_4)\,\wp\,(\mathtt{wbrb}\,n_5)$
- $\mathtt{crd} =_{def} \lambda\vec{n}.\,(\mathtt{ifrd}\,n_1)\,\wp\,(\mathtt{idrd}\,n_2)\,\wp\,(\mathtt{exrd}\,n_3)\,\wp\,(\mathtt{merd}\,n_4)\,\wp\,(\mathtt{wbrd}\,n_5)$
- $\mathtt{cwb} =_{def} \lambda\vec{n}.\,(\mathtt{ifwb}\,n_1)\,\wp\,(\mathtt{idwb}\,n_2)\,\wp\,(\mathtt{exwb}\,n_3)\,\wp\,(\mathtt{mewb}\,n_4)\,\wp\,(\mathtt{wbwb}\,n_5)$
- $\mathtt{cwd} =_{def} \lambda\vec{n}.\,(\mathtt{ifwd}\,n_1)\,\wp\,(\mathtt{idwd}\,n_2)\,\wp\,(\mathtt{exwd}\,n_3)\,\wp\,(\mathtt{mewd}\,n_4)\,\wp\,(\mathtt{wbwd}\,n_5)$

$$
\begin{array}{rcll}
\texttt{lr} & : & \texttt{int} \to \texttt{int} \to o & \\
\texttt{cond}, \texttt{mar}, \texttt{lmdr}, \texttt{smdr} & : & \texttt{int} \to o & \\
\texttt{lcond}, \texttt{lmar}, \texttt{llmdr}, \texttt{lsmdr} & : & \texttt{int} \to o & \\
\texttt{aout}, \texttt{aout}_1 & : & \texttt{int} \to o & \\
\texttt{la}, \texttt{lb}, \texttt{laout}, \texttt{laout}_1 & : & \texttt{int} \to o & \\
\texttt{pc}_1, \texttt{lpc}_\texttt{i} & : & \texttt{int} \to o & i \in \{1,2\} \\
\texttt{ir}, \texttt{lir} & : & \texttt{inst} \to o & \\
\texttt{it}_\texttt{i}, \texttt{lit}_\texttt{i} & : & \texttt{class} \to \texttt{int} \to \texttt{int} \to \texttt{func} \to o & i \in \{1,2,3\} \\
\texttt{ifrb}, \texttt{idrb}, \texttt{exrb}, \texttt{merb}, \texttt{wbrb} & : & \texttt{int} \to o & \\
\texttt{ifrd}, \texttt{idrd}, \texttt{exrd}, \texttt{merd}, \texttt{wbrd} & : & \texttt{int} \to o & \\
\texttt{ifwb}, \texttt{idwb}, \texttt{exwb}, \texttt{mewb}, \texttt{wbwb} & : & \texttt{int} \to o & \\
\texttt{ifwd}, \texttt{idwd}, \texttt{exwd}, \texttt{mewd}, \texttt{wbwd} & : & \texttt{int} \to o & \\
\texttt{alu?}, \texttt{ld?}, \texttt{st?}, \texttt{br?}, \texttt{noop?} & : & \texttt{class} \to o &
\end{array}
$$

Figure 5.9: Signature for specification of DLX

I sometimes use (`crb` $\vec{n}$) as abbreviation for (`crb` $n_1\, n_2\, n_3\, n_4\, n_5$), and similarly for `crd`, `cwb` and `cwd`. Thus (`crb` $\vec{n}$), is the signal for the read begin phase for the stages of the pipeline, and the numbers n_i , $i \in [1, \ldots, 5]$ denote different states of the signal in question. τ_1, τ_2, τ_3, τ_4 and τ_5, specified in figure 5.10, are the transition functions for IF, ID, EX, MEM and WB stages, respectively. The states of the five signals in the clock are arguments to each of the transition functions, and the output is the state of the clock signal for its stage. In figure 5.10, $n_i \in \{0, 1\}$, $i \in [3, \ldots, 5]$, and for any input not exhibited the functions τ_1, τ_2, τ_3, τ_4 and τ_5 return 1. For the ID clock signal, $i \in \{-1, -2, -3\}$ are the states for a data hazard where the stalled instruction has to wait for $-i$ cycles, $i \in \{-4, -5\}$ are the states for a control hazard, and $i = 0$ is the state when the pipeline will stop within the next five cycles. For example, from figure 5.10, $\tau_1 1(-3)100 = 0$, thus IF will remain idle in the next cycle. I sometimes use (`crb` $\tau(\vec{n})$) as an abbreviation for (`crb` $\tau_1(\vec{n})\, \tau_2(\vec{n})\, \tau_3(\vec{n})\, \tau_4(\vec{n})\, \tau_5(\vec{n})$), and similarly for `crd`, `cwb` and `cwd`.

The specification for the pipeline, $\mathcal{E}_p$, is the set of universal closures of clauses in figures 5.12, 5.13, 5.14 and 5.15. In the ID stage in figure 5.12, RAW and control hazards are detected by the function δ_l, whose return value sets the state for `idrd`. δ_l is defined below.

Definition 5.5 [δ_l - Hazard detection function for $\mathcal{E}_p$]

IF	ID	EX	MEM	WB	τ_1	τ_2	τ_3	τ_4	τ_5
1	1	1	1	1	1	1	1	1	1
1	-3	1	n_4	n_5	0	-3	0	1	n_4
0	-3	0	1	n_5	0	-3	0	0	1
0	-3	0	0	1	0	1	0	0	0
0	1	0	n_4	n_5	1	1	1	0	n_4
1	1	1	0	n_5	1	1	1	1	0
1	1	1	1	0	1	1	1	1	1
1	-2	n_3	1	n_5	0	-2	0	n_3	1
0	-2	0	n_4	1	0	1	0	0	n_4
1	-1	n_3	n_4	1	0	1	0	n_3	n_4
1	-4	n_3	n_4	n_5	0	-5	1	n_3	n_4
0	-4	0	n_4	n_5	0	-5	1	0	n_4
0	-5	1	n_4	n_5	0	-5	0	1	n_4
0	-5	0	1	n_5	1	-5	0	0	1
1	-5	0	0	1	1	1	0	0	0
1	1	0	0	0	1	1	1	0	0
1	0	n_3	n_4	n_5	-1	0	0	n_3	n_4
-1	0	0	n_4	n_5	0	0	0	0	n_5
0	0	0	0	n_5	0	0	0	0	-1

Figure 5.10: DLX pipeline state transition functions for clock

$$
\begin{aligned}
(\delta_l S_1\, S_2\, \vec{D}_i\, \vec{C}_i\, C) &= min\{(\rho_1 S_1\, \vec{D}_i\, \vec{C}_i\, C), (\rho_1 S_2\, \vec{D}_i\, \vec{C}_i\, C)\} && C \in \{\texttt{alu}, \texttt{st}\} \\
&= (\rho_1 S_1\, \vec{D}_i\, \vec{C}_i\, C) && \textit{otherwise}
\end{aligned}
$$

Where, ρ_1 is the function in figure 5.11, C is the instruction type, and S_1 and S_2 are the two source registers of the current instruction. C_i is the instruction type and D_i is the destination register of the ith preceding instruction, for $i \in \{1, 2, 3\}$. ■

Note the difference between the control hazard and data hazard. *In the case of a data hazard the instruction signaling the hazard and the ones following it are stalled. However, in the case of a control hazard, the* `br` *instruction which signals the hazard continues execution while the instructions following it are stalled.* Thus, if there is both a control and data hazard, the data hazard must be processed first. The table for ρ_1 is given in figure 5.11. In figure 5.11, $A \in \{\texttt{alu}, \texttt{br}, \texttt{ld}, \texttt{st}\}$, $Y \in \{\texttt{alu}, \texttt{br}, \texttt{ld}, \texttt{st}, \texttt{noop}, \texttt{bg}\}$, and $Z \in \{\texttt{alu}, \texttt{ld}\}$. Furthermore, $=_j$ in the column for D_i means that the destination register D_i is the source register S_j, $\neq_j$ means that the destination register D_i is different from the source register S_j, and X_j means either $=_j$ or $\neq_j$. If the current instruction is a `noop` or the destination registers of the three preceding instructions are different from the source registers, then there is no hazard. A data hazard is detected if any of the previous three instructions write either S_1 or S_2. A control hazard is signaled if the current instruction is a branch and there is no data hazard. If the current instruction is `ht`, then the pipeline is stopped after the execution of `ht` is finished.

C	C_1	C_2	C_3	D_1	D_2	D_3	δ
`noop`	Y	Y	Y	X_j	X_j	X_j	1
A	Y	Y	Y	$\neq_j$	$\neq_j$	$\neq_j$	1
A	Z	Y	Y	$=_j$	X_j	X_j	-3
A	Y	Z	Y	$\neq_j$	$=_j$	X_j	-2
A	Y	Y	Z	$\neq_j$	$\neq_j$	$=_j$	-1
`br`	Y	Y	Y	$\neq_j$	$\neq_j$	$\neq_j$	-4
`ht`	Y	Y	Y	X_j	X_j	X_j	0

Figure 5.11: ρ_1 - table for hazard detection in the DLX pipeline

The complex looking clauses in the $\mathcal{E}_p$ warrant some explanation. The clock is specified by the first two clauses in figure 5.12. It is a cyclic process consuming the completing signals for read(write) to enabling signals for write(read). The clock stops if the state for `wbwd` is -1. The state information of the clock is calculated by $\vec{\tau}$ defined in figure 5.10. Each of the five stages of the pipeline – IF, ID, EX, MEM and WB – are implemented by two clauses. One of the clauses synchronizes with the read begin signal, reads data into temporary latches, and produces the read done signal. The other clause synchronizes with the write begin signal, consumes data in the temporary latches, and produces the write done signal. The actions performed by a stage depend on the type of the instruction being processed and the state of the clock. *Note that a register might be updated in one stage and read in another stage during the same cycle.* To ensure availability of proper data, in the read phase required data is stored into latches, and only after all data is read, registers and memory may be updated in the write phase.

The `bg` instruction is handled specially as the clock is started by this instruction. This instruction also loads several latches in the environment needed by the pipeline. I use $\mathcal{L}$ as an abbreviation for the various latches as defined below, and $(\mathcal{L}\,\vec{C}\,\vec{D}\,\vec{I}\,\vec{O}\,\vec{S}\,\vec{V})$ is a *latch state*. $\mathcal{L}_0$ is the initial state for these latches. When the lengths of the vectors are not mentioned explicitly, they are of the required length, as assumed in the definition below.

$$\begin{aligned}
\mathcal{L} \quad &=_{def} \quad \lambda\vec{C}, \vec{D}, \vec{I}, \vec{O}, \vec{S}, \vec{V}.\,(\wp_{i\in\{1,2,3\}}(\mathtt{it_i}\,C_i\,D_i\,I_i\,O_i))\,\wp\,(\mathtt{ir}\,(\mathtt{ix}\;\;C_4\,S_1\,S_2\,D_4\,I_4\,O_4))\\
&\qquad \wp\,(\mathtt{a}\,V_1)\,\wp\,(\mathtt{b}\,V_2)\,\wp\,(\mathtt{aout}\,V_3)\,\wp\,(\mathtt{aout_1}\,V_4)\,\wp\,(\mathtt{pc_1}\,V_5)\,\wp\,(\mathtt{mar}\,V_6)\,\wp\\
&\qquad (\mathtt{lmdr}\,V_7)\,\wp\,(\mathtt{smdr}\,V_8)\,\wp\,(\mathtt{cond}\,V_8)\\
\mathcal{L}_0 \quad &=_{def} \quad (\mathcal{L}\;\mathtt{n\vec{o}op}\;\vec{0}\;\vec{0}\;\vec{+}\;\vec{0}\;\vec{0})
\end{aligned}$$

Now, evaluation of DLX programs can be defined using $\mathcal{E}_p$. The idea behind evaluation is the same as that for $\mathcal{E}_s$. A program is loaded in the program memory, and then the program state is evaluated in a given data state. The program state remains static, while the data state may possibly change due to the execution of instructions. The definitions of program and data state are taken from definitions 5.1 and 5.2.

Definition 5.6 [Pipelined evaluation in DLX, $\mathcal{E}_p$] Given data states $\mathcal{S}_1$ and $\mathcal{S}_2$, and $(\mathcal{P}\;l\;\vec{P_l})$, a program state. P_l evaluates in $\mathcal{S}_1$ to $\mathcal{S}_2$ written as $\mathcal{S}_1\,\wp\,(\mathcal{P}\;l\;\vec{P_l}) \mapsto_p \mathcal{S}_2\,\wp\,(\mathcal{P}\;l\;\vec{P_l})$, if

$$\Sigma_p : \mathcal{E}_p\,;\,\mathcal{S}_2\,\wp\,(\mathcal{P}\;l\;\vec{P_l}) \longrightarrow \mathcal{S}_1\,\wp\,(\mathcal{P}\;l\;\vec{P_l})$$

is provable in FORUM. ■

I use the example program in figure 5.7 to illustrate the specification $\mathcal{E}_p$. I use $\mathcal{S}_1$ as an abbreviation for the entire expression in the figure. At the end of the computation, the resultant state, $\mathcal{S}_2$ will have 9 in the second register, 4 in the `pc`, 3 in the `num` and otherwise be identical to $\mathcal{S}_1$. The proof of the computation is detailed below. Let $\mathcal{S}_1'$ be identical to $\mathcal{S}_1$, except that it does not contain `pc` and `num`, and $\mathcal{S}_1''$ be identical to $\mathcal{S}_1$ except that it does not contain `pc`, $(\mathtt{r}\,2)$ and `num`.

$$(\mathtt{crd}\ \vec{N}) \mathrel{\circ\!-} (\mathtt{cwb}\ \vec{N})$$

$$\begin{array}{l}
(\mathtt{cwd}\ \vec{N}) \mathrel{\circ\!-} \\
\quad [(\mathtt{ne}\,(N_5 = -1)) \otimes (\mathtt{crb}\ \tau(\vec{N}))] \oplus \\
\quad [(\mathtt{eq}\,(N_5 = -1)) \mathrel{\circ\!-} [\mathbf{1} \wp ((\mathcal{L}\ \vec{C}\ \vec{D}\ \vec{I}\ \vec{S}\ \vec{V}) \mathrel{\circ\!-} \bot)]]
\end{array}$$

$$\begin{array}{l}
(\mathtt{p}\ L\,(\mathtt{ix\ \ bg}\ D\,S_1\,S_2\,I\,O)) \wp (\mathtt{pc}\ L) \wp (\mathtt{num}\ M) \mathrel{\circ\!-} \\
\quad (\mathtt{p}\ L\,(\mathtt{ix\ \ bg}\ D\,S_1\,S_2\,I\,O)) \wp (\mathtt{pc}\,(L+1)) \wp (\mathtt{crb}\ \vec{1}) \wp \mathcal{L}_0 \wp (\mathtt{num}\,(m+1))
\end{array}$$

$$\begin{array}{l}
(\mathtt{ifrb}\ N) \wp (\mathtt{p}\ L\,V) \wp (\mathtt{pc}\ L) \mathrel{\circ\!-} (\mathtt{pc}\ L) \wp (\mathtt{p}\ L\,V) \wp \\
\quad [[(\mathtt{eq}\,(N = 1)) \otimes ((\mathtt{lir}\ V) \wp (\mathtt{lpc}\,(L+1)) \wp (\mathtt{ifrd}\ N))] \oplus \\
\quad\ [(\mathtt{eq}\,(N = -1)) \otimes ((\mathtt{lir}\,(\mathtt{ix\ \ noop}\ 0\,1\,2\,0\,+)) \wp (\mathtt{lpc}\,(L-1)) \wp (\mathtt{ifrd}\ N))] \oplus \\
\quad\ [(\mathtt{eq}\,(N = 0)) \otimes (\mathtt{ifrd}\ N)]]
\end{array}$$

$$\begin{array}{l}
(\mathtt{ifwb}\ N) \mathrel{\circ\!-} \\
\quad [(\mathtt{eq}\,(N = 0)) \otimes (\mathtt{ifwd}\ N)] \oplus \\
\quad\ [[(\mathtt{eq}\,(N = 1)) \oplus (\mathtt{eq}\,(N = -1))] \mathrel{\circ\!-} \\
\quad\quad [\mathbf{1} \wp ((\mathtt{ir}\ V') \wp (\mathtt{pc}\ L') \wp (\mathtt{lir}\ V) \wp (\mathtt{lpc}\ L) \mathrel{\circ\!-} (\mathtt{ir}\ V) \wp (\mathtt{pc}\ L) \wp (\mathtt{ifwd}\ N))]]
\end{array}$$

$$\begin{array}{l}
(\mathtt{idrb}\ N) \wp (\wp_{i\in\{1,2,3\}} (\mathtt{it_i}\ C_i\,D_i\,I_i\,O_i)) \wp (\mathtt{pc}\ L) \wp (\mathtt{ir}\,(\mathtt{ix}\ \ C\,D\,S_1\,S_2\,I\,O)) \mathrel{\circ\!-} \\
\quad (\wp_{i\in\{1,2,3\}} (\mathtt{it_i}\ C_i\,D_i\,I_i\,O_i)) \wp (\mathtt{pc}\ L) \wp (\mathtt{ir}\,(\mathtt{ix}\ \ C\,D\,S_1\,S_2\,I\,O)) \wp \\
\quad\ [[(\mathtt{ne}\,(N = 1)) \otimes (\mathtt{idrd}\ N)] \oplus \\
\quad\quad [(\mathtt{eq}\,(N = 1)) \otimes \\
\quad\quad\ [[[(\mathtt{eq}\,(u = 1)) \oplus (\mathtt{eq}\,(u = -4))] \otimes ((\mathtt{lit_1}\ C\,D\,I\,O) \wp (\mathtt{lpc_1}\ L)) \mathrel{\circ\!-} \\
\quad\quad\quad [[(\mathtt{r}\ S_1\,V_1) \mathrel{\circ\!-} (\mathtt{r}\ S_1\,V_1) \wp (\mathtt{la}\ V_1) \wp S] \otimes \\
\quad\quad\quad [(\mathtt{r}\ S_2\,V_2) \mathrel{\circ\!-} (\mathtt{r}\ S_2\,V_2) \wp (\mathtt{lb}\ V_2) \wp S] \otimes \\
\quad\quad\quad [S \wp S \mathrel{\circ\!-} (\mathtt{idrd}\ u)]]] \oplus \\
\quad\quad\ [(\mathtt{ne}\,(u = 1)) \otimes (\mathtt{ne}\,(u = -4)) \otimes (\mathtt{idrd}\ u)]]]] \\
\qquad\qquad \text{Where } u =_{def} (\delta_l\ S_1\,S_2\,\vec{D}_i\,\vec{C}_i\,C)
\end{array}$$

$$\begin{array}{l}
(\mathtt{idwb}\ N) \wp (\mathtt{it_1}\ C\ D\,I\,O) \mathrel{\circ\!-} \\
\quad [(\mathtt{ne}\,(N = 1)) \otimes (\mathtt{ne}\,(N = -4)) \otimes ((\mathtt{it_1}\,\mathtt{noop}\ D\,I\,O) \wp (\mathtt{idwd}\ N))] \oplus \\
\quad [[(\mathtt{eq}\,(N = 1)) \oplus (\mathtt{eq}\,(N = -4))] \mathrel{\circ\!-} \\
\quad\ [[(\mathtt{a}\ W_1) \wp (\mathtt{la}\ V_1) \wp (\mathtt{b}\ W_2) \wp (\mathtt{lb}\ V_2) \wp (\mathtt{pc_1}\ L_1) \wp (\mathtt{lpc_1}\ L_2) \wp (\mathtt{lit_1}\ C'\,D'\,I'\,O') \mathrel{\circ\!-} \\
\quad\quad (\mathtt{a}\ V_1) \wp (\mathtt{b}\ V_2) \wp (\mathtt{pc_1}\ L_2) \wp (\mathtt{it_1}\ C'\,D'\,I'\,O') \wp (\mathtt{idwd}\ N) \wp \mathbf{1}]]]
\end{array}$$

Figure 5.12: Specification for the DLX pipeline – clock, bg, IF and ID.

$(\texttt{alu?}\,\texttt{alu}) \multimap \mathbf{1}$

$(\texttt{ld?}\,\texttt{ld}) \multimap \mathbf{1}$

$(\texttt{st?}\,\texttt{st}) \multimap \mathbf{1}$

$(\texttt{br?}\,\texttt{br}) \multimap \mathbf{1}$

$(\texttt{noop?}\,\texttt{noop}) \multimap \mathbf{1}$

$$
\begin{array}{l}
(\texttt{exrb}\,N) \wp (\texttt{it}_1\,C\,D\,I\,O) \multimap (\texttt{it}_1\,C\,D\,I\,O) \wp (\texttt{lit}_2\,C\,D\,I\,O) \wp \\
\quad [[(\texttt{ne}\,(N = 1)) \otimes (\texttt{exrd}\,N)] \oplus \\
\quad [(\texttt{eq}\,(N = 1)) \otimes \\
\quad\; [(\texttt{alu?}\,C) \multimap \\
\quad\quad ((\texttt{a}\,V_1) \wp (\texttt{b}\,V_2) \multimap (\texttt{a}\,V_1) \wp (\texttt{b}\,V_2) \wp (\texttt{laout}\,(V_1\;O\;V_2)) \wp (\texttt{exrd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{ld?}\,C) \multimap \\
\quad\quad ((\texttt{a}\,V_1) \multimap (\texttt{a}\,V_1) \wp (\texttt{lmar}\,(V_1 + I)) \wp (\texttt{exrd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{st?}\,C) \multimap \\
\quad\quad ((\texttt{a}\,V_1) \wp (\texttt{b}\,V_2) \multimap \\
\quad\quad\quad (\texttt{a}\,V_1) \wp (\texttt{b}\,V_2) \wp (\texttt{lmar}\,(V_1 + I)) \wp (\texttt{lsmdr}\,V_2) \wp (\texttt{exrd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{br?}\,C) \multimap \\
\quad\quad ((\texttt{pc}_1\,L) \wp (\texttt{a}\,V_1) \multimap \\
\quad\quad\quad (\texttt{pc}_1\,L) \wp (\texttt{a}\,V_1) \wp (\texttt{laout}\,(L + I)) \wp (\texttt{lcond}\,(V_1\;O\;0)) \wp (\texttt{exrd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{noop?}\,C) \otimes (\texttt{exrd}\,N)]]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{exwb}\,N) \wp (\texttt{it}_2\,C'\,D'\,I'\,O') \wp (\texttt{lit}_2\,C\,D\,I\,O) \multimap (\texttt{it}_2\,C\,D\,I\,O) \wp \\
\quad [[(\texttt{ne}\,(N = 1)) \otimes (\texttt{exwd}\,N)] \oplus \\
\quad [(\texttt{eq}\,(N = 1)) \otimes \\
\quad\; [(\texttt{alu?}\,C) \multimap \\
\quad\quad ((\texttt{aout}\,V_1) \wp (\texttt{laout}\,V_2) \multimap (\texttt{aout}\,V_2) \wp (\texttt{exwd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{ld?}\,C) \multimap \\
\quad\quad ((\texttt{mar}\,L_1) \wp (\texttt{lmar}\,L_2) \multimap (\texttt{mar}\,L_2) \wp (\texttt{exwd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{st?}\,C) \multimap \\
\quad\quad ((\texttt{mar}\,L_1) \wp (\texttt{lmar}\,L_2) \wp (\texttt{smdr}\,W_1) \wp (\texttt{lsmdr}\,W_2) \multimap \\
\quad\quad\quad (\texttt{mar}\,L_2) \wp (\texttt{smdr}\,W_2) \wp (\texttt{exwd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{br?}\,C) \multimap \\
\quad\quad ((\texttt{aout}\,V_1) \wp (\texttt{laout}\,V_2) \wp (\texttt{cond}\,U_1) \wp (\texttt{lcond}\,U_2) \multimap \\
\quad\quad\quad (\texttt{aout}\,V_2) \wp (\texttt{cond}\,U_2) \wp (\texttt{exwd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{noop?}\,C) \otimes (\texttt{exwd}\,N)]]]
\end{array}
$$

Figure 5.13: Specification for the DLX pipeline – EX.

$$
\begin{array}{l}
(\texttt{merb}\,N)\wp(\texttt{it}_2\,C\,D\,I\,O) \multimap (\texttt{it}_2\,C\,D\,I\,O)\wp(\texttt{lit}_3\,C\,D\,I\,O)\wp \\
\quad [(\texttt{ne}\,(N=1)) \otimes (\texttt{merd}\,N)]\oplus \\
\quad [(\texttt{eq}\,(N=1))\otimes \\
\quad\;\; [(\texttt{alu?}\,C) \multimap \\
\quad\quad ((\texttt{aout}\,V_1) \multimap (\texttt{aout}\,V_1)\wp(\texttt{laout}_1\,V_1)\wp(\texttt{merd}\,N))\wp\mathbf{1}]\oplus \\
\quad\;\; [(\texttt{ld?}\,C) \multimap \\
\quad\quad ((\texttt{mar}\,L)\wp(\texttt{m}\,L\,V) \multimap \\
\quad\quad\quad (\texttt{mar}\,L)\wp(\texttt{m}\,L\,V)\wp(\texttt{llmdr}\,V)\wp(\texttt{merd}\,N))\wp\mathbf{1}]\oplus \\
\quad\;\; [(\texttt{st?}\,C) \multimap \\
\quad\quad ((\texttt{mar}\,L)\wp(\texttt{m}\,L\,V)\wp(\texttt{smdr}\,V_2) \multimap \\
\quad\quad\quad (\texttt{mar}\,L)\wp(\texttt{m}\,L\,V_2)\wp(\texttt{smdr}\,V_2)\wp(\texttt{merd}\,N))\wp\mathbf{1}]\oplus \\
\quad\;\; [(\texttt{br?}\,C) \multimap \\
\quad\quad ((\texttt{cond}\,M)\wp(\texttt{aout}\,V_1)\wp(\texttt{pc}\,L_2) \multimap \\
\quad\quad\quad [[(\texttt{ne}\,(M=1)) \otimes ((\texttt{lpc}_2\,L_2)\wp(\texttt{merd}\,N))]\oplus \\
\quad\quad\quad\quad [(\texttt{eq}\,(M=1)) \otimes ((\texttt{lpc}_2\,V_1)\wp(\texttt{merd}\,N))]] \\
\quad\quad\quad \wp(\texttt{cond}\,M)\wp(\texttt{aout}\,L_1)\wp(\texttt{pc}\,L_2))\wp\mathbf{1}]\oplus \\
\quad\;\; [(\texttt{noop?}\,C) \otimes (\texttt{merd}\,N)]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{mewb}\,N)\wp(\texttt{it}_3\,C'\,D'\,I'\,O')\wp(\texttt{lit}_3\,C\,D\,I\,O) \multimap (\texttt{it}_3\,C\,D\,I\,O)\wp \\
\quad [(\texttt{ne}\,(N=1)) \multimap (\texttt{mewd}\,N)\wp\mathbf{1}]\oplus \\
\quad [(\texttt{eq}\,(N=1))\otimes \\
\quad\;\; [(\texttt{alu?}\,C) \multimap \\
\quad\quad ((\texttt{aout}_1\,V_1)\wp(\texttt{laout}_1\,V_2) \multimap (\texttt{aout}_1\,V_2)\wp(\texttt{mewd}\,N))\wp\mathbf{1}]\oplus \\
\quad\;\; [(\texttt{ld?}\,C) \multimap \\
\quad\quad ((\texttt{lmdr}\,W_1)\wp(\texttt{llmdr}\,W_2) \multimap (\texttt{lmdr}\,W_2)\wp(\texttt{mewd}\,N))\wp\mathbf{1}]\oplus \\
\quad\;\; [[(\texttt{st?}\,C)\oplus(\texttt{noop?}\,C)] \otimes (\texttt{mewd}\,N)]\oplus \\
\quad\;\; [(\texttt{br?}\,C) \multimap \\
\quad\quad ((\texttt{pc}\,L_1)\wp(\texttt{lpc}_2\,L_2) \multimap (\texttt{pc}\,L_2)\wp(\texttt{mewd}\,N))\wp\mathbf{1}]]
\end{array}
$$

Figure 5.14: Specification for the DLX pipeline – MEM.

$$
\begin{array}{l}
(\texttt{wbrb}\, N) \wp (\texttt{it}_3\, C\, D\, I\, O) \wp (\texttt{num}\, M) \mathrel{\circ\!-} (\texttt{it}_3\, C\, D\, I\, O) \wp \\
\quad [(\texttt{eq}\,(N = 0)) \otimes ((\texttt{wbrd}\, N) \wp (\texttt{num}\, M))] \oplus \\
\quad [(\texttt{eq}\,(N = -1)) \otimes ((\texttt{wbrd}\, N) \wp (\texttt{num}\,(M + 1)))] \oplus \\
\quad [(\texttt{eq}\,(N = 1)) \otimes \\
\quad [(\texttt{alu?}\, C) \mathrel{\circ\!-} \\
\qquad ((\texttt{aout}_1\, V_1) \mathrel{\circ\!-} (\texttt{aout}_1\, V_1) \wp (\texttt{lr}\, 1\, d\, V_1) \wp (\texttt{wbrd}\, N) \wp (\texttt{num}\,(M + 1))) \wp \mathbf{1}] \oplus \\
\quad [(\texttt{ld?}\, C) \mathrel{\circ\!-} \\
\qquad ((\texttt{lmdr}\, V_1) \mathrel{\circ\!-} (\texttt{lmdr}\, V_1) \wp (\texttt{lr}\, 1\, d\, V_1) \wp (\texttt{wbrd}\, N) \wp (\texttt{num}\,(M + 1))) \wp \mathbf{1}] \oplus \\
\quad [((\texttt{st?}\, C) \oplus (\texttt{br?}\, C)) \otimes ((\texttt{lr}\, 0\, d\, V_1) \wp (\texttt{wbrd}\, N) \wp (\texttt{num}\,(M + 1)))] \oplus \\
\quad [(\texttt{noop?}\, C) \otimes ((\texttt{lr}\, 0\, d\, V_1) \wp (\texttt{wbrd}\, N) \wp (\texttt{num}\, M))]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{wbwb}\, N) \mathrel{\circ\!-} \\
\quad [(\texttt{ne}\,(N = 1)) \otimes (\texttt{wbwd}\, N)] \oplus \\
\quad [(\texttt{eq}\,(N = 1)) \mathrel{\circ\!-} \\
\qquad [[(\texttt{lr}\, 1\, d\, V_1) \wp (\texttt{r}\, d\, V_2) \mathrel{\circ\!-} (\texttt{r}\, d\, V_1) \wp (\texttt{wbwd}\, N)]\ \& \\
\qquad [(\texttt{lr}\, 0\, d\, V_1) \mathrel{\circ\!-} (\texttt{wbwd}\, N)] \wp \mathbf{1}]]
\end{array}
$$

Figure 5.15: Specification for the DLX pipeline – WB.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\sigma_1}{\mathcal{S}_2 \longrightarrow (\texttt{ifrd}\, 1) \wp (\texttt{ir}\, X_1) \wp (\texttt{pc}\, 3) \wp \Gamma_3}}{\mathcal{S}_2 \longrightarrow (\texttt{cwb}\, \vec{1}) \wp (\texttt{lir}\, X_1) \wp (\texttt{lpc}\, 3) \wp \Gamma_2}\ \texttt{ifwb}}{\mathcal{S}_2 \longrightarrow (\texttt{crd}\, \vec{1}) \wp (\texttt{lir}\, X_1) \wp (\texttt{lpc}\, 3) \wp \Gamma_2}\ \mathit{Clock}}{\texttt{idrb},\ \texttt{exrb}\ \vdots\ \texttt{merb},\ \texttt{wbrb}}}{\mathcal{S}_2 \longrightarrow (\texttt{ifrd}\, 1) \wp (\texttt{lir}\, X_1) \wp (\texttt{lpc}\, 3) \wp \Gamma_1}}{\dfrac{\mathcal{S}_2 \longrightarrow (\texttt{pc}\, 2) \wp \mathcal{S}_1' \wp (\texttt{crb}\, \vec{1}) \wp \mathcal{L}_0 \wp (\texttt{num}\, 1)}{\Sigma_p : \mathcal{E}_s ; \mathcal{S}_2 \longrightarrow \mathcal{S}_1}\ \texttt{bg}}\ \texttt{ifrb}
$$

The proof begins by backchaining on the `bg` clause. Each of the `it` registers has a `noop` instruction (by the definition of $\mathcal{L}_0$). Thus, only the execution of `ifrb` and `ifwb` stages are shown. Note that there is no hazard detected because the instruction in $\texttt{it}_1$ was a `noop`. X_1 is the instruction ($\texttt{ix}\ \ \texttt{alu}\, 2\, 3\, 2\, I +$), and X_2 is the instruction ($\texttt{ix}\ \ \texttt{ht}\, 1\, 2\, 3\, 0 +$). Now I complete σ_1 below.

$$\dfrac{\dfrac{\sigma_2}{\begin{array}{ll} S \wp S \multimap (\mathtt{idrd}\,1) & \\ (\mathtt{r}\,3\,5) \multimap (\mathtt{r}\,3\,5) \wp (\mathtt{b}\,5) \wp S, & (\mathtt{ifrd}\,1) \wp (\mathtt{lir}\,X_2) \wp (\mathtt{lpc}\,4) \wp (\mathtt{lpc_1}\,3) \\ \mathcal{S}_2, (\mathtt{r}\,2\,4) \multimap (\mathtt{r}\,2\,4) \wp (\mathtt{a}\,4) \wp S \longrightarrow & \wp (\mathtt{lit_1}\,\mathtt{alu}\,2\,0\,+) \wp \Gamma_5 \end{array}}}{}$$

$$\dfrac{\dfrac{\dfrac{\mathtt{ifrb},\ \mathtt{idrb} \ \vdots\ \textit{other logical rule}}{\mathcal{S}_2 \longrightarrow (\mathtt{crb}\ \vec{1}) \wp (\mathtt{ir}\,X_1) \wp (\mathtt{pc}\,3) \wp \Gamma_4}\ \mathtt{ifrb}}{\mathcal{S}_2 \longrightarrow (\mathtt{cwd}\ \vec{1}) \wp (\mathtt{ir}\,X_1) \wp (\mathtt{pc}\,3) \wp \Gamma_4}\ \textit{Clock}}{}$$

$$\dfrac{\mathtt{idwb},\ \mathtt{exwb} \ \vdots\ \mathtt{mewb},\ \mathtt{wbwb}}{\mathcal{S}_2 \longrightarrow (\mathtt{ifrd}\,1) \wp (\mathtt{ir}\,X_1) \wp (\mathtt{pc}\,3) \wp \Gamma_3}$$

I use Γ_i in the proofs above for the part of the context which is inactive in the rule. The construction of σ_2 is similar to the plan above. First the ID stage is completed for X_1, the EX, MEM and WB have `noop` instructions. In the next cycle, `ht` is detected and the state of `idwd` becomes 0, X_1 proceeds to EX stage. Now the pipeline goes through the final stages as the clock shuts down. The construction of σ_2 is not made explicit because it gets very tedious and the idea is clear from the above constructions.

Note that in the read and the write phase I choose a given order for backtracking on the various clauses. *It is a key fact that the order in which the various read clauses are chosen for backtracking during the read phase is irrelevant to provability, and similarly for the write phase* – a similar point was made in the context of process theories in [Mil93]. The reason for this "permutability" in the read phase is that information is read by the various stages into different latches, and in the write phase the stages use the latches they "created" to update the values of distinct storage locations. In other words, there is no contention for resources in a given phase, read or write, amongst the various stages. *It is this permutability of backtracking which underlines the concurrent nature of the pipeline specification in FORUM.*

Now I have $\mathcal{E}_s$ and $\mathcal{E}_p$, two operational semantics for DLX. On the one hand, $\mathcal{E}_s$ is much easier to work with and understand, and on the other hand, $\mathcal{E}_p$ specifies a much more interesting algorithm. The natural question to ask is whether $\mathcal{E}_s$ and $\mathcal{E}_p$ are equivalent. Following is the Correspondence theorem that I would like to prove.

Theorem 5.7 (Correspondence between $\mathcal{E}_s$ and $\mathcal{E}_p$) *For any* DLX *program P_l, and* *data states* $\mathcal{S}_1$ *and* $\mathcal{S}_2$,

$$\mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_s \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l) \quad \text{iff} \quad \mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_p \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l)$$

The theorem will be proved by induction on the number of instructions executed. Since the internal states of the two specifications are different, the theorem is obtained as a corollary to the lemma 5.9 which establishes relationships between the sequential and pipelined evaluation using the different internal states. Before I present these lemmas, I prove some properties about the evaluation using $\mathcal{E}_p$. I define `it` to be consistent if $(\mathtt{it}_1\, C_1\, D_1\, I_1\, O_1)$ is the instruction in EX stage, $(\mathtt{it}_2\, C_2\, D_2\, I_2\, O_2)$ is the instruction in MEM stage, and $(\mathtt{it}_3\, C_3\, D_3\, I_3\, O_3)$ is the instruction in WB stage. The actual proofs, which are constructions of proofs in FORUM, are not shown in the proofs of the lemmas in this section, essentially because the construction gets very tedious and unilluminating. However, the steps outlined provide the recipe for constructing the required proofs.

Lemma 5.8 ($\mathcal{E}_p$ - `it` consistency, data hazards, and control hazards) *If*

- $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *are two latch states, and* `it` *is consistent in* $\mathcal{L}_1$,
- $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are two data states, such that the* `pc` *in* $\mathcal{S}_1$ *does not address a* `bg` *or* `ht` *instruction in* $\mathcal{P}$,
- $\mathcal{P}$ *is a program state, and*
- $\Sigma_p : \mathcal{E}_p\,;\, \mathcal{S}_2 \wp \mathcal{L}_2 \wp (\mathtt{cwd}\, \vec{N}) \wp \mathcal{P} \vdash \mathcal{S}_1 \wp \mathcal{L}_1 \wp (\mathtt{crb}\, \vec{N}) \wp \mathcal{P}$,

then

1. `it` *is consistent in* $\mathcal{L}_2$,
2. *if the instruction in the* ID *stage depends upon a preceding instruction in the pipeline for data, then the data hazard is resolved, and*

3. if the instruction in the ID *stage is a* br*, then the control hazard is resolved.*

Proof: Part 1 : For this part, notice that in every completion of a read and write phase $\texttt{it}_3$ gets the values of $\texttt{it}_2$, which in turn gets the values from $\texttt{it}_1$, which in turn get the values from ir in the ID. This movement of values of it exactly matches the definition of the consistency of it.

Part 2 : Given the consistency of it, δ_l returns one of -1, -2 or -3, *i.e.* a data hazard, if and only if the instruction in ID stage depends upon one of the preceding instructions for data. From the definition of the state transition function in figure 5.10, it is clear that the instruction with the dependency and the ones following that instruction are stalled till the preceding instruction completes execution.

Part 3 : Given the consistency of it, δ_l detects returns -4, *i.e.* a control hazard, if and only if the current instruction is a br and there are no data hazards. The absence of data hazards enables the execution of br. From figure 5.10, the pipeline after the br is stalled till the br completes the MEM stage, when the new pc is available. Thus the instruction after a br is fetched from the right address. ■

The problem in relating the computations of $\mathcal{E}_s$ and $\mathcal{E}_p$ are manifold. Firstly, the number of steps taken to compute a program are different - $\mathcal{E}_p$ takes five steps for each instruction - hence an induction on the height of the proof trees would not work. Secondly, the internal states of the two evaluators are different. *The induction measure that I use is the number of instructions which have completed evaluation. However, I have to treat the case when the last instruction is* ht *separately from other instructions because it cleans up the environment of the latch state.*

Lemma 5.9 (Lemmas for Correspondence between $\mathcal{E}_s$ and $\mathcal{E}_p$) *Given*

- $m \geq 1$ *and* $m \in \mathsf{nat}$,

- $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *two data states minus the* pc, num *has* 0 *and* m *in* $\mathcal{S}_1$ *and* $\mathcal{S}_2$, *respectively,*

- P_l *is a* DLX *program,* $\mathcal{P}_1 =_{def} (\mathcal{P}\ l\ \vec{P_l})$ *is a program state,*

- $\mathcal{L}_1 =_{def} \forall \vec{C}, \vec{D}, \vec{I}, \vec{O}, \vec{S}, \vec{V}. (\mathcal{L}\, \vec{C}\, \vec{D}\, \vec{I}\, \vec{O}\, \vec{S}\, \vec{V})$, *and*

- $\mathcal{C} =_{def} \forall \vec{N}. (\mathtt{cwd}\, \vec{N})$.

1. *if the mth instruction is not* ht *then*

$$\Sigma_s : \mathcal{E}_s ; \mathcal{S}_2 \wp \, \mathtt{cont} \wp (\mathtt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1 \quad \textit{iff}$$
$$\Sigma_p : \mathcal{E}_p ; \mathcal{S}_2 \wp (\mathtt{pc}\, K) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1$$

2. *if the mth instruction is* ht *then*

$$\Sigma_s : \mathcal{E}_s ; \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1 \quad \textit{iff}$$
$$\Sigma_p : \mathcal{E}_p ; \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1$$

Proof: Proof Part 1 (Left to Right) : The m_{th} instruction is not ht.

We have

$$\Sigma_s : \mathcal{E}_s ; \mathcal{S}_2 \wp \, \mathtt{cont} \wp (\mathtt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1.$$

Suppose the m_{th} instruction is a bg. Then there are two cases :

$m = 1$: Then bg is the first instruction, $\mathcal{S}_2$ and $\mathcal{S}_1$ are identical except for the count in num, and the proof is completed by following the execution of the bg instruction.

$m = n + 1, 0 < n$: Then bg is not the first instruction. By the grammar for programs (figure 5.5), the n_{th} instruction executed must be a ht. Thus $\mathcal{S}_2'$, the state at the end of the n_{th} instruction, is the same as $\mathcal{S}_2$ except for the count in `num`. Using induction on Part 2, I get

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2' \wp (\mathtt{pc}\,(L-1)) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1.$$

The shape of the right-hand side of the sequent at the point when `num` is incremented to n will be

$$\mathcal{S}_2' \wp (\mathtt{pc}\,(L-1)) \wp \mathcal{P}_1.$$

Backchaining on the clause for `bg` increments `num` by 1 and the right-hand side of the sequent becomes

$$\mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{L}_0 \wp (\mathtt{crb}\, \vec{0}) \wp \mathcal{P}_1.$$

Now in the linear part of the left-hand-side of the sequent, I replace

$$\mathcal{S}_2' \wp (\mathtt{pc}\,(L-1)) \wp \mathcal{P}_1 \quad \text{by} \quad \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1.$$

Note that this replacement does not effect the structure of the proof thus far, and the sequent that I am now constructing the proof of is

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\mathtt{pc}\, M) \wp \mathcal{P}_1.$$

After the transformation all that I have left to prove is

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1 \vdash \mathcal{S}_2 \wp (\mathtt{pc}\, L) \wp \mathcal{L}_0 \wp (\mathtt{crb}\, \vec{0}) \wp \mathcal{P}_1.$$

The proof is easily completed by using $\wp - L$, $\forall - L$ and Id, and hence I am done.

Suppose the m_{th} instruction is not a bg. Then the definition of the DLX programs implies that $m = n + 1$, $0 < n$ and that the n_{th} instruction cannot be ht. By induction hypothesis on Part 1, I get

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}'_2 \wp \mathcal{L}_1 \wp \mathcal{C} \wp (\texttt{pc}\, K) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\texttt{pc}\, M) \wp \mathcal{P}_1.$$

$\mathcal{S}'_2$ differs from $\mathcal{S}_2$ in that it has n in num and is missing the side effects, if any, of the m_{th} instruction.

First consider the case when the n_{th} instruction is not a branch instruction. The computation up to the execution of the n_{th} instruction matches for $\mathcal{E}_s$ and $\mathcal{E}_p$. Note that the n_{th} instruction is at $(L - 2)$ address in the program memory. By lemma 5.8 parts 1 and 2, it is consistent, and data-dependency, if any, for the m_{th} instruction is resolved. Thus, the arguments to the m_{th} instruction are identical to the sequential computation, and hence, so are the results. At the point when num is incremented to n the right-hand side of the sequent has the following shape

$$\mathcal{S}'_2 \wp (\texttt{pc}\, K') \wp (\mathcal{L}\, \vec{V}) \wp (\texttt{crb}\, \vec{W}) \wp \mathcal{P}_1.$$

Where K' depends upon the data and control hazards encountered by the instructions following the n_{th} instruction. For example, if there are no hazards then K' will be $L + 3$. Backchaining over clauses to complete the execution of the m_{th} instruction results in a right-hand side that looks like

$$\mathcal{S}_2 \wp (\texttt{pc}\, K) \wp (\mathcal{L}\, \vec{V}') \wp (\texttt{crb}\, \vec{W}') \wp \mathcal{P}_1.$$

To complete the proof, in the linear part of the left-hand-side of the sequent, replace

$$\mathcal{S}'_2 \wp \mathcal{L}_1 \wp \mathcal{C} \wp (\texttt{pc}\, K) \wp \mathcal{P}_1 \text{ by } \mathcal{S}_2 \wp \mathcal{L}_1 \wp \mathcal{C} \wp (\texttt{pc}\, K) \wp \mathcal{P}_1.$$

The transformation does not effect the structure of the proof, and as a result of it I am now constructing the proof for the sequent

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2 \wp \mathcal{L}_1 \wp \mathcal{C} \wp (\texttt{pc}\, K) \wp \mathcal{P}_1 \longrightarrow \mathcal{S}_1 \wp (\texttt{pc}\, M) \wp \mathcal{P}_1.$$

Given the right-hand side above, the proof can be finished by $\wp - L$, $\forall - L$ and Id, and hence I am done.

Now consider the case when the n_{th} instruction is a branch instruction. Firstly, the argument to `br` is identical to the one in the sequential run, and thus by lemma 5.8 part 3, the next `pc` calculated at the end of the MEM stage will also be $(L-1)$. Again, using part 2 of lemma 5.8, the data dependency, if any, for the m_{th} instruction is resolved, and thus the result produced by the m_{th} instruction are identical to the ones produced in the sequential run. By the definition of the transition functions in figure 5.10, at the end of the cycle when `num` is incremented to n the right-hand side of the sequent has the following shape

$$\mathcal{S}_2' \wp (\texttt{pc}\, L) \wp (\mathcal{L}\, \vec{V}) \wp (\texttt{crb}\, \vec{W}) \wp \mathcal{P}_1$$

and the m_{th} instruction will have completed its IF stage. The rest of the argument can be completed analogous to the case above when the n_{th} instruction was not `br`.

Proof Part 2 (Left to Right) :

We have

$$\Sigma_s : \mathcal{E}_s \,;\, \mathcal{S}_2 \wp (\texttt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\texttt{pc}\, M) \wp \mathcal{P}_1.$$

The m_{th} instruction to be executed - `ht` - is at the $(L-1)$ address in the program memory. By the grammar for DLX programs (figure 5.5) there has to be at least one instruction that was completed before `ht`, and the immediately preceding instruction could not be another ht. Thus $m = n + 1, 0 < n$. The state $\mathcal{S}_2'$ at the end of the n_{th} instruction is the same as $\mathcal{S}_2$, except that num stores n.

By induction hypotheses on part 1, I have

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2' \wp (\texttt{pc}\, K) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\texttt{pc}\, M) \wp \mathcal{P}_1.$$

First consider the case when the n_{th} instruction is not a branch instruction. Note that the n_{th} instruction is at the $(L-2)$ address in the program memory. In the proof I look at the point when the n_{th} instruction completed its ID phase. At this point the right-hand side of the sequent will have the following shape,

$$\mathcal{S}_2' \wp (\mathcal{L}\, \vec{V}) \wp (\texttt{pc}\, L) \wp (\texttt{cwd}\, \vec{W}) \wp \mathcal{P}_1.$$

Now, in this proof at the point ht completed the ID stage, the state of the idrd was set to 0, and pc has $(L+1)$. τ_1 sets the state for ifrb is set to -1 which causes the pc to be decremented by 1 by the IF stage, and thus pc has L. Now, from the definition of the state transition functions it is clear that the stages of the pipeline become idle as the ht passes through them. WB knows that ht has arrived when the state of wbrb is -1, and it increases the count in num without looking at the class of the instruction in $\texttt{it}_3$. Now the clock will detect the -1 as its fifth argument and it will consume the $(\mathcal{L}\, \vec{V}')$ from the environment, and I am left with $\mathcal{S}_2 \wp (\texttt{pc}\, L) \wp \mathcal{P}_1$.

In the proof thus far, in the linear part of the left-hand-side of the sequent, I replace

$$\mathcal{S}_2' \wp (\texttt{pc}\, K) \wp \mathcal{L}_1 \wp \mathcal{C} \wp \mathcal{P}_1 \text{ by } \mathcal{S}_2 \wp (\texttt{pc}\, L) \wp \mathcal{P}_1.$$

This transformation does not effect the structure of the proof thus far, and now I am proving

$$\Sigma_p : \mathcal{E}_p \,;\, \mathcal{S}_2 \wp (\texttt{pc}\, L) \wp \mathcal{P}_1 \vdash \mathcal{S}_1 \wp (\texttt{pc}\, M) \wp \mathcal{P}_1.$$

As a result of the transformation the proof can be completed immediately using an identity axiom.

Now consider the case when the n_{th} instruction is a branch instruction. Firstly, the argument to `br` is identical to the one in the sequential run, and thus by lemma 5.8 part 3, the next `pc` calculated at the end of the MEM will also be $(L-1)$. By the definition of the transition functions in figure 5.10, at the end of the cycle when `num` is incremented to n the right-hand side of the sequent has the following shape

$$\mathcal{S}_2' \wp (\mathtt{pc}\, L) \wp (\mathcal{L}\, \vec{V}) \wp (\mathtt{crd}\vec{W}) \wp \mathcal{P}_1$$

and the m_{th} instruction will have completed its IF stage. The rest of the argument can be completed analogous to the case above when the n_{th} instruction was not `br`.

The proofs going from right to left can be completed along similar lines - the proofs for $\mathcal{E}_s$ are composed using linear cuts. ■

The proof highlights the main fact that the key ingredient in the specification of the pipeline is the complex synchronization and hazard detection. The lemma 5.9 is a inductive argument using the basic properties of the $\mathcal{E}_p$ as proved in lemma 5.8. The proof is rather straight forward, given the choice of induction measure, however, the complete construction of FORUM proofs is rather cumbersome. The nice feature is that the two declarative specifications - $\mathcal{E}_s$ and $\mathcal{E}_p$ - were proved equivalent, and cuts were used in one direction of the proof.

5.4 Call-forwarding and early branch resolution

The specification $\mathcal{E}_s$ exhibits the basic ideas of the pipeline. However, the pipeline deals naively with the data and control hazards it faces - it stalls whenever it may need to. In this section I look at – call forwarding, early branch resolution, branch prediction – simple and important techniques to reduce the stalls caused by data dependencies and branch instructions.

Suppose an instruction, X_1, in the ID stage needs the result of an `alu` instruction which is in

CONTROL

IF	$\mathsf{IR} := \mathsf{M}[\mathsf{PC}];\ \mathsf{PC} := \mathsf{PC} + 1$
ID	$\mathsf{A} := \mathsf{Rs1};\ \mathsf{B} := \mathsf{Rs2};\ \mathsf{PC}_1 := \mathsf{PC};\ \mathsf{IR}_1 := \mathsf{IR}$ $\mathsf{BTA} := \mathsf{PC} + (\mathsf{IR}_{16})^{16} \sharp\sharp \mathsf{IR}_{16 \ldots 31}$ $\mathsf{if}\ (\mathsf{Rs1}\ \mathsf{op}\ 0)\ \{\mathsf{PC} := \mathsf{BTA}\}$
EX	
MEM	
WB	

Figure 5.16: Changes in the DLX pipeline to reduce branch penalty.

the MEM stage. According to the scheme of $\mathcal{E}_p$, X_1 has to stall for `alu` to go through MEM and WB before it can get the result of `alu` – this is clearly inefficient. The rather straightforward idea of short-circuiting the loop, *i.e.* "forwarding" the result from `aout` directly to the input of X_1, works very well in practise. Sending results from one functional units in the pipeline to another functional unit directly is called *call forwarding*. Call forwarding reduces dramatically the stalls generated in the pipeline due to data dependencies.

The second inefficiency of $\mathcal{E}_p$ is regarding the cost of branch instructions. The idea here is to calculate the outcome of the branch instruction as early as possible in the pipeline – early branch resolution. The basic change to the flow for branch instructions is shown in figure 5.16. By using dedicated adders, extra latches and other circuitry it is possible to compute the new `pc` by the end of the ID stage. This reduces the penalty for branches from three cycles to one cycle. The other aspect of optimization for branches is to first predict whether a branch will be taken or not, and then continue to fetch and execute from the predicted address till the branch is resolved – branch prediction. In case the branch alters the `pc`, the earlier instructions are invalidated, and the execution starts at the new `pc`. In case the branch does not alter the `pc`, the machine has incurred no penalty by continuing to compute rather than sit idly as $\mathcal{E}_p$ does. The number of instructions computed speculatively will never be more than one in the DLX pipeline, because the new `pc` is ready at the end of the ID stage of the branch instruction.

Call forwarding and speculative computation become more critical to the performance of the pipeline in the presence of instructions which take more than one cycle to complete. Extensions to, and variants of these ideas are embodied in many of present day RISC machines [HP90]. The specification of these features require more complex synchronizations, new definitions of the state transition functions, and call forwarding functions. Further, the IF and ID stages will have to synchronize directly with each other. Consider the situation when a branch instruction is in the ID stage. The branch instruction might alter the `pc` at the same time when IF wants to increment the old value in the `pc`. Thus, there is a race for the `pc`, and the final value in the `pc` is unpredictable. I force the write phase of IF to start after the write phase of the ID stage, and thus only IF writes to the `pc`, avoiding the race.

I begin by presenting the definition for the new hazard resolution function, δ_o, for detecting hazards given that I am implementing call-forwarding and early branch resolution. The table for ρ_2 is given in figure 5.17. In figure 5.17, $A \in \{\texttt{alu}, \texttt{br}, \texttt{ld}, \texttt{st}\}$, and $Y \in \{\texttt{alu}, \texttt{br}, \texttt{ld}, \texttt{st}, \texttt{noop}, \texttt{bg}\}$. Furthermore, $=_j$ in the column for D_i means that the destination register D_i is the source register S_j, $\neq_j$ means that the destination register D_i is different from the source register S_j, and X_j means either $=_j$ or $\neq_j$. *The first point to note is that δ_o only depends upon the two immediately preceding instructions in the pipeline, not three as in the case of δ_l.* Note, moreover, the difference between the `alu` and `ld` instructions. The result of the `alu` is available at the end of the EX stage, while that of the `ld` is available only after the MEM stage. Thus, a dependency with an instruction two cycles ahead of the current one is signaled only when the instruction two cycle ahead is a `ld`. Thus the cost of data hazard with `alu` instructions is reduced to one cycle and with `ld` instructions to two cycles at most. Other than this, ρ_2 is same as ρ_1.

Using the definition of ρ_2, I define below δ_o, the hazard detection function, for $\mathcal{E}_o$, the new specification of pipeline incorporating call-forwarding and early branch resolution.

Definition 5.10 [δ_o - Hazard detection function for $\mathcal{E}_o$]

C	C_1	C_2	D_1	D_2	δ_o
`noop`	Y	Y	X	X	1
A	Y	Y	$\neq_j$	$\neq_j$	1
A	`ld`	Y	$=_j$	X	-2
A	Y	`ld`	$\neq_j$	$=_j$	-1
A	`alu`	Y	$=_j$	X	-1
`br`	Y	Y	$\neq_j$	$\neq_j$	-4
`ht`	Y	Y	X	X	0

Figure 5.17: ρ_2 – new table for hazard detection in DLX pipeline

$$\begin{aligned}(\delta_o S_1\, S_2\, \vec{D}_i\, \vec{C}_i\, C) &= min\{(\rho_2 S_1\, \vec{D}_i\, \vec{C}_i\, C), (\rho_2 S_2\, \vec{D}_i\, \vec{C}_i\, C)\} && C \in \{\texttt{alu}, \texttt{st}\} \\ &= (\rho_2 S_1\, \vec{D}_i\, \vec{C}_i\, C) && otherwise\end{aligned}$$

Where, ρ_2 is the function in figure 5.17. C is the instruction type, and S_1 and S_2 are the two source registers of the current instruction. C_i is the instruction type, and D_i is the destination register of the ith preceding instruction, for $i \in \{1, 2\}$. ■

To make the pipeline work with call-forwarding and early branch resolution, I need to redefine the state transition functions for the clock. The main point is that fewer stall signals are generated when either data or control hazards are detected. σ_1, σ_2, σ_3, σ_4 and σ_5, specified in figure 5.18, are the new transition functions for IF, ID, EX, MEM and WB stages, respectively. The states of the five signals in the clock are arguments to each of the transition functions, and the output is the state of the clock signal for its stage. In figure 5.18, $n_i \in \{-1, 0, 1\}$, $i \in [3, \ldots, 5]$, and for any input not exhibited the functions σ_1, σ_2, σ_3, σ_4 and σ_5 return 1. For the ID clock signal, $i \in \{-1, -2\}$ are the states for a data hazard where the stalled instruction has to wait for $-i$ cycles, $i \in \{-4, -5\}$ are the states for a control hazard, and $i = 0$ is the state when the pipeline will stop within the next five cycles. For example, from figure 5.18, $\sigma_1 1(-2)100 = 0$, thus IF will remain idle in the next cycle. I sometimes use ($\texttt{crb}$ $\sigma(\vec{n})$) as an abbreviation for ($\texttt{crb}$ $\sigma_1(\vec{n})\, \sigma_2(\vec{n})\, \sigma_3(\vec{n})\, \sigma_4(\vec{n})\, \sigma_5(\vec{n})$), and similarly for `crd`, `cwb` and `cwd`.

Other than changing the definitions of the transition functions and hazard detection, I need to know from which unit a value has to be forwarded to which unit. In the case of the DLX

IF	ID	EX	MEM	WB	σ_1	σ_2	σ_3	σ_4	σ_5
1	1	1	1	1	1	1	1	1	1
1	-2	1	n_4	n_5	0	-2	0	1	n_4
0	-2	0	n_4	n_5	0	1	0	0	n_4
0	1	0	n_4	n_5	1	1	1	0	n_4
1	1	1	0	n_5	1	1	1	1	0
1	1	1	1	0	1	1	1	1	1
1	-1	n_3	1	n_5	0	1	0	n_3	1
1	-4	n_3	n_4	n_5	1	-5	1	n_3	n_4
0	-4	0	n_4	n_5	1	-5	1	0	n_4
1	-5	1	n_4	n_5	1	1	-1	1	n_4
1	1	-1	1	n_5	1	1	1	-1	1
1	1	1	-1	1	1	1	1	1	0
1	0	n_3	n_4	n_5	-1	0	0	n_3	n_4
-1	0	0	n_4	n_5	0	0	0	0	n_4
0	0	0	0	n_5	0	0	0	0	-1

Figure 5.18: DLX pipeline state transition functions in the presence of call-forwarding and early branch resolution

architecture that I am studying, all data is read in the ID stage, and thus the destination of the forwarded data will be one of the input latches to the ALU. φ_1 and φ_2, defined in figure 5.19, are the two call-forwarding functions for the two source registers of a given instruction. The result of a `ld` is forwarded only when C_3 has `ld`, because by this time the instruction has completed its MEM stage. In the case of `alu` instructions, results are available after the ID stage, and the result is forwarded when `alu` result is either C_2 or C_3. Note that just as ρ_2 does not need $\texttt{it}_3$, the call-forwarding functions do not need $\texttt{it}_1$. *It is interesting to note that the two tables for ρ_2 and call-forwarding functions put together cover all the cases in the table for ρ_1.* The notation in figure 5.19 is the same as that in figure 5.17. The numeric return values signal the register from where the data will come - 0 stands for the register file, 1 stands for `lmdr`, 2 stands for `aout` and 3 stands for $\texttt{aout}_1$.

The specification for the pipeline, $\mathcal{E}_o$, is the set of universal closures of clauses in figures 5.20, 5.21 and 5.22. The signature for the specification Σ_o is the union of Σ_p and $\{\texttt{bta} : int \rightarrow o\}$. Note that the branch is resolved at the end of ID stage, and hence the EX

C	C_2	C_3	D_2	D_3	φ_1	$\varphi_2(\texttt{br},\texttt{ld})$
A	Y	$\neq_j$	$\neq_j$	0	0	
A	Y	`ld`	$\neq_j$	$=_j$	1	1(0)
A	`alu`	Y	$=_j$	X	2	2(0)
A	Y	`alu`	$\neq_j$	$=_j$	3	3(0)

Figure 5.19: φ – Call forwarding functions

and MEM stages treat the branch instruction as if it were a `noop`. Other than this change, the clauses for EX and MEM are identical to the clauses for these stages in $\mathcal{E}_p$. The clause for the clock, `bg` instruction, `eq`, `ne`, WB stage and for matching classes remain unchanged, but I have rewritten them here for the sake of completeness. The critical difference is in the clauses for IF and ID.

The read phase of ID has to take into account call-forwarding and calculation of the jump address for the branch in `bta`. Note that the calculation of the jump address is done for all instructions, because at this point the class of the instruction has not been decoded. If ID is not idle then first hazards are checked. If there is a data hazard the instruction stalls. Otherwise, if there is either no hazard or a control hazard then the current instruction will continue. The call forwarding functions are used to obtain the arguments of the instruction from the appropriate registers. Note that the call forwarding functions are used only when there are no data hazards. Furthermore, if a control hazard is detected then the state of `idrd` will be -4.

If there were no hazards in the ID stage, the state of `idwb` is 1, `bta` is consumed away and appropriate registers are loaded. Note that the register $\texttt{pc}_1$ is no longer needed because this was used only by `br` in the EX stage. The case remaining is when the state of `idwb` is -4, *i.e.* the instruction in ID is a branch instruction. If the condition of the branch instruction is true, then `bta` is left in the environment and the state of the `idwd` is -4. If, however, the condition of the branch instruction is false then `bta` is consumed and the state of `idwd` is set to 1.

The write phase of IF stage must synchronize with `idwd` – this ensures that ID has already finished its write phase. If the state of `idwd` is not -4, then the instruction in ID did not alter the `pc`, and the actions of IF are identical to the write phase of IF in $\mathcal{E}_p$. In case state of `idwd` is -4, `bta` is used to set the value of `pc`. The read phase of IF is unchanged from $\mathcal{E}_p$.

The idea behind evaluation remains the same as for $\mathcal{E}_p$. The definition of data and program state are from definition5.1 and 5.2. The definition of latch state is different in that the latch state for $\mathcal{E}_o$ does not have the registers `cond` and $\texttt{pc}_1$. Having said this, I will use the same notation for the latch state for $\mathcal{E}_o$ also.

Definition 5.11 [Pipelined evaluation in DLX, $\mathcal{E}_o$] Given data states $\mathcal{S}_1$ and $\mathcal{S}_2$, and $(\mathcal{P}\ l\ \vec{P}_l)$, a program state. P_l evaluates in $\mathcal{S}_1$ to $\mathcal{S}_2$ written as $\mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_o \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l)$, if

$$\Sigma_o : \mathcal{E}_o\, ;\, \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l) \longrightarrow \mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l)$$

is provable in FORUM. ■

I have $\mathcal{E}_s$ and $\mathcal{E}_o$, two operational semantics for DLX. I will prove the equivalence of these two specifications along the lines of the proof for theorem 5.7.

Theorem 5.12 (Correspondence between $\mathcal{E}_s$ and $\mathcal{E}_o$) *For any* DLX *program P_l, and data states $\mathcal{S}_1$ and $\mathcal{S}_2$,*

$$\mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_s \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l) \quad \text{iff} \quad \mathcal{S}_1 \wp (\mathcal{P}\ l\ \vec{P}_l) \mapsto_o \mathcal{S}_2 \wp (\mathcal{P}\ l\ \vec{P}_l)$$

Before proving the theorem, the following properties of the $\mathcal{E}_o$ will be proved. These lemmas prove that $\mathcal{E}_o$ maintains the consistency of `it`, and resolves data and control hazards correctly. Analogous facts were proved for $\mathcal{E}_p$ in lemma 5.8.

Lemma 5.13 ($\mathcal{E}_o$ - `it` consistency, data hazards, and control hazards) *If*

$$
\begin{array}{l}
(\mathtt{ifrb}\,N)\wp(\mathtt{p}\,L\,V)\wp(\mathtt{pc}\,L) \multimap (\mathtt{pc}\,L)\wp(\mathtt{p}\,L\,V)\wp \\
\quad [[(\mathtt{eq}\,(N=1)) \otimes ((\mathtt{lir}\,V)\wp(\mathtt{lpc}\,(L+1))\wp(\mathtt{ifrd}\,N))]\oplus \\
\quad\ [(\mathtt{eq}\,(N=-1)) \otimes ((\mathtt{lir}\,(\mathtt{ix}\ \mathtt{noop}\,0\,1\,2\,0+))\wp(\mathtt{lpc}\,(L-1))\wp(\mathtt{ifrd}\,N))]\oplus \\
\quad\ [(\mathtt{eq}\,(N=0)) \otimes (\mathtt{ifrd}\,N)]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{ifwb}\,N)\wp(\mathtt{idwd}\,M) \multimap (\mathtt{idwd}\,M)\wp \\
\quad [[((\mathtt{eq}\,(N=0)) \otimes (\mathtt{ne}\,(M=-4))) \otimes (\mathtt{ifwd}\,N)]\oplus \\
\quad\ [((\mathtt{eq}\,(N=0)) \otimes (\mathtt{eq}\,(M=-4))) \multimap \\
\qquad ((\mathtt{bta}\,L'')\wp(\mathtt{pc}\,L') \multimap (\mathtt{pc}\,L'')\wp(\mathtt{ifwd}\,N))\wp\mathbf{1}]\oplus \\
\quad\ [((\mathtt{eq}\,(N=1)) \otimes (\mathtt{eq}\,(M=-4))) \multimap \\
\qquad ((\mathtt{ir}\,V')\wp(\mathtt{pc}\,L')\wp(\mathtt{lir}\,V)\wp(\mathtt{lpc}\,L)\wp(\mathtt{bta}\,L'') \multimap \\
\qquad\quad (\mathtt{ir}\,V)\wp(\mathtt{pc}\,L'')\wp(\mathtt{ifwd}\,N))\wp\mathbf{1}]\oplus \\
\quad\ [(\mathtt{eq}\,(N=-1))\oplus((\mathtt{eq}\,(N=1)) \otimes (\mathtt{ne}\,(M=-4))) \multimap \\
\qquad ((\mathtt{ir}\,V')\wp(\mathtt{pc}\,L')\wp(\mathtt{lir}\,V)\wp(\mathtt{lpc}\,L) \multimap (\mathtt{ir}\,V)\wp(\mathtt{pc}\,L)\wp(\mathtt{ifwd}\,N))\wp\mathbf{1}]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{idrb}\,N)\wp(\wp_{i\in\{1,2,3\}}(\mathtt{it_i}\,C_i\,D_i\,I_i\,O_i))\wp(\mathtt{pc}\,L)\wp(\mathtt{ir}\,(\mathtt{ix}\ \ C\,D\,S_1\,S_2\,I\,O)) \multimap \\
\quad (\wp_{i\in\{1,2,3\}}(\mathtt{it_i}\,C_i\,D_i\,I_i\,O_i))\wp(\mathtt{pc}\,L)\wp(\mathtt{ir}\,(\mathtt{ix}\ \ C\,D\,S_1\,S_2\,I\,O))\wp \\
\quad\ [[(\mathtt{ne}\,(N=1)) \otimes (\mathtt{idrd}\,N)]\oplus \\
\quad\ \ [(\mathtt{eq}\,(N=1))\otimes \\
\qquad [[((\mathtt{eq}\,(u=1))\oplus((\mathtt{eq}\,(u=-4)) \otimes (\mathtt{bta}\,(L+I)))) \otimes (\mathtt{lit_1}\,C\,D\,I\,O) \multimap \\
\qquad\ [(\mathtt{arg_r}\,f_1\,S_1\,\mathtt{la}\,V_1) \otimes (\mathtt{arg_r}\,f_2\,S_2\,\mathtt{lb}\,V_2) \otimes [S\wp S \multimap (\mathtt{idrd}\,u)]]]\oplus \\
\qquad [(\mathtt{ne}\,(u=1)) \otimes (\mathtt{ne}\,(u=-4)) \otimes (\mathtt{idrd}\,u)]]]] \\
u =_{def} (\delta_o\,S_1\,S_2\,D_1\,D_2\,C_1\,C_2\,C), \qquad f_i =_{def} (\varphi_1\,S_i\,D_2\,D_3\,C_2\,C_3\,C) \qquad\qquad i\in[1,2] \\
\mathtt{arg_r} =_{def} \lambda\,u,\,s,\,l,\,v.[[(\mathtt{eq}\,(0=u))\wp(\mathtt{r}\,s\,v) \multimap (\mathtt{r}\,s\,v)\wp(l\,v)]\ \& \\
\qquad\qquad\qquad\qquad\quad [(\mathtt{eq}\,(1=u))\wp(\mathtt{lmdr}\,v) \multimap (\mathtt{lmdr}\,v)\wp(l\,v)]\ \& \\
\qquad\qquad\qquad\qquad\quad [(\mathtt{eq}\,(2=u))\wp(\mathtt{aout}\,v) \multimap (\mathtt{aout}\,v)\wp(l\,v)]\ \& \\
\qquad\qquad\qquad\qquad\quad [(\mathtt{eq}\,(3=u))\wp(\mathtt{aout_1}\,v) \multimap (\mathtt{aout_1}\,v)\wp(l\,v)]]
\end{array}
$$

$$
\begin{array}{l}
(\mathtt{idwb}\,N)\wp(\mathtt{it_1}\,C\,D\,I\,O) \multimap \\
\quad [(\mathtt{ne}\,(N=1)) \otimes (\mathtt{ne}\,(N=-4)) \otimes ((\mathtt{it_1}\,\mathtt{noop}\,D\,I\,O)\wp(\mathtt{idwd}\,N))]\oplus \\
\quad [(\mathtt{eq}\,(N=1)) \otimes (\mathtt{bta}\,L) \multimap \\
\qquad [(\mathtt{a}\,W_1)\wp(\mathtt{la}\,V_1)\wp(\mathtt{b}\,W_2)\wp(\mathtt{lb}\,V_2)\wp(\mathtt{lit_1}\,C'\,D'\,I'\,O') \multimap \\
\qquad\quad (\mathtt{a}\,V_1)\wp(\mathtt{b}\,V_2)\wp(\mathtt{it_1}\,C'\,D'\,I'\,O')\wp(\mathtt{idwd}\,N)]]\oplus \\
\quad [(\mathtt{eq}\,(N=-4)) \otimes (\mathtt{bta}\,L) \multimap \\
\qquad [(\mathtt{a}\,W_1)\wp(\mathtt{la}\,V_1)\wp(\mathtt{b}\,W_2)\wp(\mathtt{lb}\,V_2)\wp(\mathtt{lit_1}\,C'\,D'\,I'\,O') \multimap \\
\qquad\quad (\mathtt{a}\,V_1)\wp(\mathtt{b}\,V_2)\wp(\mathtt{it_1}\,C'\,D'\,I'\,O')\wp \\
\qquad\quad [[(\mathtt{eq}\,(V_1\,O'\,0)) \otimes ((\mathtt{bta}\,L)\wp(\mathtt{idwd}\,N))]\oplus[(\mathtt{ne}\,(V_1\,O'\,0)) \otimes (\mathtt{idwd}\,1)]]]]
\end{array}
$$

Figure 5.20: Specification for the DLX pipeline – IF and ID.

$$(\texttt{crd}\ \vec{N}) \multimap (\texttt{cwb}\ \vec{N})$$

$$(\texttt{cwd}\ \vec{N}) \multimap$$
$$[(\texttt{ne}\,(N_5 = -1)) \otimes (\texttt{crb}\ \tau(\vec{N}))] \oplus [(\texttt{eq}\,(N_5 = -1)) \multimap [\mathbf{1} \wp ((\mathcal{L}\ \vec{C}\ \vec{D}\ \vec{I}\ \vec{S}\ \vec{V}) \multimap \bot)]]$$

$$(\texttt{p}\ L\,(\texttt{ix bg}\ D\ S_1\ S_2\ I\ O)) \wp (\texttt{pc}\ L) \multimap$$
$$(\texttt{p}\ L\,(\texttt{ix bg}\ D\ S_1\ S_2\ I\ O)) \wp (\texttt{pc}\,(L+1)) \wp (\texttt{crb}\ \vec{1}) \wp \mathcal{L}_0$$

$$(\texttt{alu?}\ \texttt{alu}) \multimap \mathbf{1} \qquad (\texttt{ld?}\ \texttt{ld}) \multimap \mathbf{1}$$

$$(\texttt{st?}\ \texttt{st}) \multimap \mathbf{1} \qquad (\texttt{br?}\ \texttt{br}) \multimap \mathbf{1}$$

$$(\texttt{noop?}\ \texttt{noop}) \multimap \mathbf{1}$$

$$(\texttt{exrb}\ N) \wp (\texttt{it}_1\ C\ D\ I\ O) \multimap (\texttt{it}_1\ C\ D\ I\ O) \wp (\texttt{lit}_2\ C\ D\ I\ O)$$
$$[(\texttt{ne}\,(N=1)) \otimes (\texttt{exrd}\ N)] \oplus$$
$$[(\texttt{eq}\,(N=1)) \otimes$$
$$[(\texttt{alu?}\ C) \multimap$$
$$((\texttt{a}\ V_1) \wp (\texttt{b}\ V_2) \multimap (\texttt{a}\ V_1) \wp (\texttt{b}\ V_2) \wp (\texttt{laout}\,(V_1\ O\ V_2)) \wp (\texttt{exrd}\ N)) \wp \mathbf{1}] \oplus$$
$$[(\texttt{ld?}\ C) \multimap$$
$$((\texttt{a}\ V_1) \multimap (\texttt{a}\ V_1) \wp (\texttt{lmar}\,(V_1 + I)) \wp (\texttt{exrd}\ N)) \wp \mathbf{1}] \oplus$$
$$[(\texttt{st?}\ C) \multimap$$
$$((\texttt{a}\ V_1) \wp (\texttt{b}\ V_2) \multimap$$
$$(\texttt{a}\ V_1) \wp (\texttt{b}\ V_2) \wp (\texttt{lmar}\,(V_1 + I)) \wp (\texttt{lsmdr}\ V_2) \wp (\texttt{exrd}\ N)) \wp \mathbf{1}] \oplus$$
$$[((\texttt{noop?}\ C) \oplus (\texttt{br?}\ C)) \otimes (\texttt{exrd}\ N)]]$$

$$(\texttt{exwb}\ N) \wp (\texttt{it}_2\ C'\ D'\ I'\ O') \wp (\texttt{lit}_2\ C\ D\ I\ O) \multimap (\texttt{it}_2\ C\ D\ I\ O)$$
$$[(\texttt{ne}\,(N=1)) \otimes (\texttt{exwd}\ N)]] \oplus$$
$$[(\texttt{eq}\,(N=1)) \otimes$$
$$[(\texttt{alu?}\ C) \multimap$$
$$((\texttt{aout}\ V_1) \wp (\texttt{laout}\ V_2) \multimap (\texttt{aout}\ V_2) \wp (\texttt{exwd}\ N)) \wp \mathbf{1}] \oplus$$
$$[(\texttt{ld?}\ C) \multimap$$
$$((\texttt{mar}\ L_1) \wp (\texttt{lmar}\ L_2) \multimap (\texttt{mar}\ L_2) \wp (\texttt{exwd}\ N)) \wp \mathbf{1}] \oplus$$
$$[(\texttt{st?}\ C) \multimap$$
$$((\texttt{mar}\ L_1) \wp (\texttt{lmar}\ L_2) \wp (\texttt{smdr}\ W_1) \wp (\texttt{lsmdr}\ W_2) \multimap$$
$$(\texttt{mar}\ L_2) \wp (\texttt{smdr}\ W_2) \wp (\texttt{exwd}\ N)) \wp \mathbf{1}] \oplus$$
$$[((\texttt{noop?}\ C) \oplus (\texttt{br?}\ C)) \otimes (\texttt{exwd}\ N)]]$$

Figure 5.21: Specification for the DLX pipeline – clock, bg, EX.

$$
\begin{array}{l}
(\texttt{merb}\,N) \wp (\texttt{it}_2\,C\,D\,I\,O) \multimap (\texttt{it}_2\,C\,D\,I\,O) \wp (\texttt{lit}_3\,C\,D\,I\,O) \wp \\
\quad [(\texttt{ne}\,(N=1)) \otimes (\texttt{merd}\,N)] \oplus \\
\quad [(\texttt{eq}\,(N=1)) \otimes \\
\quad\; [(\texttt{alu?}\,C) \multimap ((\texttt{aout}\,V_1) \multimap (\texttt{aout}\,V_1) \wp (\texttt{laout}_1\,V_1) \wp (\texttt{merd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{ld?}\,C) \multimap \\
\quad\;\; ((\texttt{mar}\,L) \wp (\texttt{m}\,L\,V) \multimap (\texttt{mar}\,L) \wp (\texttt{m}\,L\,V) \wp (\texttt{llmdr}\,V) \wp (\texttt{merd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{st?}\,C) \multimap \\
\quad\;\; ((\texttt{mar}\,L) \wp (\texttt{m}\,L\,V) \wp (\texttt{smdr}\,V_2) \multimap \\
\quad\quad\quad (\texttt{mar}\,L) \wp (\texttt{m}\,L\,V_2) \wp (\texttt{smdr}\,V_2) \wp (\texttt{merd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [((\texttt{noop?}\,C) \oplus (\texttt{br?}\,C)) \otimes (\texttt{merd}\,N)]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{mewb}\,N) \wp (\texttt{it}_3\,C'\,D'\,I'\,O') \wp (\texttt{lit}_3\,C\,D\,I\,O) \multimap (\texttt{it}_3\,C\,D\,I\,O) \wp \\
\quad [(\texttt{ne}\,(N=1)) \otimes (\texttt{mewd}\,N)] \oplus \\
\quad [(\texttt{eq}\,(N=1)) \otimes \\
\quad\; [(\texttt{alu?}\,C) \multimap ((\texttt{aout}_1\,V_1) \wp (\texttt{laout}_1\,V_2) \multimap (\texttt{aout}_1\,V_2) \wp (\texttt{mewd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{ld?}\,C) \multimap ((\texttt{lmdr}\,W_1) \wp (\texttt{llmdr}\,W_2) \multimap (\texttt{lmdr}\,W_2) \wp (\texttt{mewd}\,N)) \wp \mathbf{1}] \oplus \\
\quad\; [((\texttt{st?}\,C) \oplus (\texttt{noop?}\,C) \oplus (\texttt{br?}\,C)) \otimes (\texttt{mewd}\,N)]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{wbrb}\,N) \wp (\texttt{it}_3\,C\,D\,I\,O) \wp (\texttt{num}\,M) \multimap (\texttt{it}_3\,C\,D\,I\,O) \wp \\
\quad [(\texttt{eq}\,(N=0)) \otimes ((\texttt{wbrd}\,N) \wp (\texttt{num}\,M))] \oplus \\
\quad [(\texttt{eq}\,(N=-1)) \otimes ((\texttt{wbrd}\,N) \wp (\texttt{num}(M+1)))] \oplus \\
\quad [(\texttt{eq}\,(N=1)) \otimes \\
\quad\; [(\texttt{alu?}\,C) \multimap \\
\quad\;\; ((\texttt{aout}_1\,V_1) \multimap (\texttt{aout}_1\,V_1) \wp (\texttt{lr}\,1\,d\,V_1) \wp (\texttt{wbrd}\,N) \wp (\texttt{num}(M+1))) \wp \mathbf{1}] \oplus \\
\quad\; [(\texttt{ld?}\,C) \multimap \\
\quad\;\; ((\texttt{lmdr}\,V_1) \multimap (\texttt{lmdr}\,V_1) \wp (\texttt{lr}\,1\,d\,V_1) \wp (\texttt{wbrd}\,N) \wp (\texttt{num}(M+1))) \wp \mathbf{1}] \oplus \\
\quad\; [((\texttt{st?}\,C) \oplus (\texttt{br?}\,C)) \otimes ((\texttt{lr}\,0\,d\,V_1) \wp (\texttt{wbrd}\,N) \wp (\texttt{num}(M+1)))] \oplus \\
\quad\; [(\texttt{noop?}\,C) \otimes ((\texttt{lr}\,0\,d\,V_1) \wp (\texttt{wbrd}\,N) \wp (\texttt{num}\,M))]]
\end{array}
$$

$$
\begin{array}{l}
(\texttt{wbwb}\,N) \multimap \\
\quad [(\texttt{ne}\,(N=1)) \otimes (\texttt{wbwd}\,N)] \oplus \\
\quad [(\texttt{eq}\,(N=1)) \multimap \\
\quad\;\; [[(\texttt{lr}\,1\,d\,V_1) \wp (\texttt{r}\,d\,V_2) \multimap (\texttt{r}\,d\,V_1) \wp (\texttt{wbwd}\,N)]\ \& \\
\quad\;\; [(\texttt{lr}\,0\,d\,V_1) \multimap (\texttt{wbwd}\,N)] \wp \mathbf{1}]]
\end{array}
$$

Figure 5.22: Specification for the DLX pipeline – MEM and WB.

- *$\mathcal{L}_1$ and $\mathcal{L}_2$ are two latch states, and* `it` *is consistent in $\mathcal{L}_1$,*

- *$\mathcal{S}_1$ and $\mathcal{S}_2$ are two data states, such that the* `pc` *in $\mathcal{S}_1$ does not address a* `bg` *or* `ht` *instruction in $\mathcal{P}$,*

- *$\mathcal{P}$ is a program state, and*

- $\Sigma_o : \mathcal{E}_o \,;\, \mathcal{S}_2 \wp \mathcal{L}_2 \wp (\texttt{cwd}\, \vec{N}) \wp \mathcal{P} \vdash \mathcal{S}_1 \wp \mathcal{L}_1 \wp (\texttt{crb}\, \vec{N}) \wp \mathcal{P}$,

then

1. `it` *is consistent in $\mathcal{L}_2$,*

2. *if the instruction in the* ID *stage depends upon a preceding instruction in the pipeline for data, then the data hazard is resolved, and*

3. *if the instruction in the* ID *stage is a* `br` *then the control hazard is resolved.*

Proof: Identical to proof of lemma 5.8 using the definitions of δ_o and call forwarding functions. ■

The following lemma is proved by mutual induction on the number of instructions which have completed execution. Since the proof of lemma 5.13 is identical to the proof of lemma 5.9, I state the lemma without proof. The proof of theorem 5.12 is a corollary of this lemma.

Lemma 5.14 (Lemmas for Correspondence between $\mathcal{E}_s$ and $\mathcal{E}_o$) *Given*

- *$m \geq 1$ and $m \in$* nat*,*

- *$\mathcal{S}_1$ and $\mathcal{S}_2$ two data states minus the* `pc`, `num` *has 0 and m in $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively,*

- *P_l is a* DLX *program, $\mathcal{P}_1 =_{def} (\mathcal{P}\; l\; \vec{P_l})$ is a program state,*

- *$\mathcal{L}_1 =_{def} \forall \vec{C}, \vec{D}, \vec{I}, \vec{O}, \vec{S}, \vec{V}. (\mathcal{L}\, \vec{C}\, \vec{D}\, \vec{I}\, \vec{O}\, \vec{S} \vec{V})$, and*

- $\mathcal{C} \;=_{def} \forall \vec{N}.\,(\texttt{cwd}\,\vec{N})$.

1. *if the mth instruction is not* ht *then*

$$\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_2 \wp\, \texttt{cont} \wp\, (\texttt{pc}\, L) \wp\, \mathcal{P}_1 \vdash \mathcal{S}_1 \wp\, (\texttt{pc}\, M) \wp\, \mathcal{P}_1 \quad \textit{iff}$$
$$\Sigma_o : \mathcal{E}_o\,;\, \mathcal{S}_2 \wp\, (\texttt{pc}\, K) \wp\, \mathcal{L}_1 \wp\, \mathcal{C} \wp\, \mathcal{P}_1 \vdash \mathcal{S}_1 \wp\, (\texttt{pc}\, M) \wp\, \mathcal{P}_1$$

2. *if the mth instruction is* ht *then*

$$\Sigma_s : \mathcal{E}_s\,;\, \mathcal{S}_2 \wp\, (\texttt{pc}\, L) \wp\, \mathcal{P}_1 \vdash \mathcal{S}_1 \wp\, (\texttt{pc}\, M) \wp\, \mathcal{P}_1 \quad \textit{iff}$$
$$\Sigma_o : \mathcal{E}_o\,;\, \mathcal{S}_2 \wp\, (\texttt{pc}\, L) \wp\, \mathcal{P}_1 \vdash \mathcal{S}_1 \wp\, (\texttt{pc}\, M) \wp\, \mathcal{P}_1$$

The technique of using external functions for state transition functions, and using the logic to manage the synchronizations has provided a powerful and flexible tool. The specification for the optimizations did not change the global structure of the specification. The key redefinitions were in the external functions, and the clauses for IF and ID stages. The change from $\mathcal{E}_p$ to $\mathcal{E}_o$ is not modular, in fact, I believe it cannot be modular because we are changing the interpreter for DLX programs, not adding new constructs to the programming language. In spite of the non-modular changes the structure of the specification is maintained, and the proof strategies for the $\mathcal{E}_p$ suffice for $\mathcal{E}_o$.

5.5 Program equivalence for DLX — Correctness of code scheduling

In this section I study the observational equivalence for DLX programs. Two programs are deemed to be observationally equivalent if the *observable behavior* of the two programs is identical with respect to a given set of environments. The problem of deciding when two code fragments are observationally equivalent is of great importance to compiler optimizations. For example, code rescheduling [HP90], *i.e.* reordering the instructions in the program, is one of the most important techniques to reduce penalties due to data and control hazards. However, code can be reordered only if the reordered code is observationally equivalent to the original code sequence. Similarly, other optimizations done by back end compilers need to be justified by proving appropriate observational equivalence.

The two key words in the informal definition of observational equivalence are environment and observable behavior. The evaluator for DLX programs translates a given data state into another one. Thus, the observable entity at the end of the computation is the data state. Note, however, that if I observe the entire data state then I will be able to count the number of instructions executed by a program. Although the number of instructions executed to compute a result is useful information, this notion of equivalence would be too fine. I am primarily interested in making sure that the results computed by two programs are identical in all environments – how many steps are taken to achieve the results is a question that I am not investigating here. The result of the computation is to alter the contents of the register file and the memory – the observable entities. Hence, two programs will be equivalent if, when placed in identical environment, the contents of the register file and memory at the end of the computations is identical. The observable state of a computation is defined to be the registers, the number of memory cells and the contents of the memory.

Definition 5.15 [Observable State] $(\mathtt{r}\,1\,)\ldots(\mathtt{r}\,32\,)$ are the DLX registers, and $n \in \mathsf{nat}$ is the number of memory cells. Let $\mathcal{O}$ be an abbreviation for

$$\lambda n, \vec{V}, \vec{U}.\,(\mathtt{r}\,1\,V_1)\,\wp\,\ldots\,\wp\,(\mathtt{r}\,32\,V_{32})\,\wp\,(\mathtt{m}\,1\,U_1)\,\wp\,\ldots(\mathtt{m}\,n\,U_n).$$

The lengths of $\vec{V}$ and $\vec{U}$ – 32 and n respectively – if implicit, are assumed to be of appropriate length.

For any $n \geq 0$, $V_1, \ldots, V_{32}, U_1, \ldots, U_n\ :\ \mathtt{int}$, $(\mathcal{O}\ n\,\vec{V}\,\vec{U})$ *is an observable state.* ■

The next problem is to define the notion of environment. An environment, written as $E[]_{l,m}$, is defined to be a DLX program, P_{l+m}, in which at most one of the lists parsed by the non-terminal $H_{m,m}$ is missing. A *block* is a list of DLX instructions which can be parsed by $H_{m,m}$. Note that given a block Q of length q, replacing the hole in the environment $E[]_{l,m}$ by Q, written as $E[Q]_{l,q}$, results in a DLX program P_{l+q}. Thus, blocks of different lengths may be substituted for the hole in any given environment. I will drop the subscripts on the environments whenever this will not create any confusion. Note that I will use DLX programs themselves to test equivalence of DLX blocks. Using these two concepts, the definition of observational equivalence is made below.

Definition 5.16 [Observational Equivalence, $\cong_{dlx}$] Let Q_i, $i \in [1, 2]$ be blocks. Q_1 is observationally equivalent to Q_2, written as $Q_1 \cong_{dlx} Q_2$, if for any DLX environment E and observable states $\mathcal{O}_1$ and $\mathcal{O}_2$, the following is provable.

$$(\texttt{pc}\, L_1) \wp (\texttt{num}(N_1 + M_1)) \wp \mathcal{O}_2 \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \wp (\texttt{num}\, N_1)$$
$$\text{if and only if}$$
$$(\texttt{pc}\, L_2) \wp (\texttt{num}(N_2 + M_2)) \wp \mathcal{O}_2 \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \wp (\texttt{num}\, N_2)$$
$$\text{for some } N_i,\ M_i,\ L_i \text{ and } l_i \text{ is the length of } E[Q_i],\ i \in [1, 2].$$

■

Note that the definition is with respect to $\mathcal{E}_s$. This is enough due to the theorems 5.12 and 5.7, which establish that all three, $\mathcal{E}_p$, $\mathcal{E}_o$ and $\mathcal{E}_s$, are equivalent. This fact is very helpful because $\mathcal{E}_s$ is the simplest to work with. The definition may appear rather weak because it is only regarding observational equivalence of blocks. This is only apparent because blocks are a basic entity for which it makes sense to formalize the notion of equivalence. The key property of a block as defined is that it can only be entered at the beginning and exited at the end. If either of these two conditions are violated, then it would be next to impossible to find interesting equivalences. Suppose that the definition of a block was such that it allowed one to enter it at some intermediate point. In this case, the standard code rescheduling would be incorrect. To make this point concrete, lets look at the following example. Let their be two instructions ($\texttt{ix alu}\, 1\, 2\, 3\, I{+}$) and ($\texttt{ix alu}\, 4\, 5\, 6\, I{+}$). The first one writes the sum of registers 2 and 3 in register 1, and the second one writes the sum of registers 5 and 6 in register 4. It is clear that interchanging the order of these two instructions is harmless, *only if both the instructions are executed.* Suppose I was allowed to place the instructions in an environment which can jump to the second instruction avoiding the execution of (ix alu 1 2 3 I+), then the reordering of the instructions will produce different results, *i.e.* code rescheduling in this case would be unsound. The definition of environments and blocks disallows this possibility because the environment cannot jump to an intermediate point in a block. I now prove the observational equivalence for the general statement of code rescheduling.

Lemma 5.17 (Observational Equivalence of Code rescheduling) *Let*
$Q_1 =_{def} Ix_1; Ix_2$, $Q_2 =_{def} Ix_2; Ix_1$, *where* $Ix_i =_{def}$ $(\texttt{ix}\ \ C_i\, D_i\, S_i\, R_i\, I_i\, O_i)$, $C_i \in \{\texttt{alu}, \texttt{ld}\,\texttt{st}\}$, *and all of* D_i, S_i, R_i *are pair wise distinct for* $i \in [1, 2]$.
$Q_1 \cong_{dlx} Q_2$

Proof: Note that both Q_1 and Q_2 are blocks by definition. let E be any environment, and $\mathcal{O}_1$ and $\mathcal{O}_2$ be two observable states such that

$$(\texttt{pc}\, L_1) \wp (\texttt{num}(N_1 + M_1)) \wp \mathcal{O}_2 \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \wp (\texttt{num}\, N_1).$$

If the computation does not reach Q_1, it will also not reach Q_2, and thus the proof for

$$(\texttt{pc}\, L_2) \wp (\texttt{num}(N_2 + M_2)) \wp \mathcal{O}_2 \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \wp (\texttt{num}\, N_2)$$

is the proof that I assumed.

So suppose that

$$(\texttt{pc}\, L_3) \wp (\texttt{num}(N_3 + M_3)) \wp \mathcal{O}_2' \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \wp (\texttt{num}\, N_1)$$

such that Ix_1 is at the address L_3 in the program memory. As the block Q_1 has not been executed as yet, I must have,

$$(\texttt{pc}\, L_4) \wp (\texttt{num}(N_4 + M_4)) \wp \mathcal{O}_2' \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \mapsto_s \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \wp (\texttt{num}\, N_2)$$

such that Ix_2 is at the address L_4 in the program memory.

Computing the instruction Ix_1 and Ix_2 in both the proofs will yield

$$(\texttt{pc}\, L_3 + 2) \wp (\texttt{num}(N_3 + M_3 + 2)) \wp \mathcal{O}_2'' \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \mapsto_s \\ \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_1\, E[\vec{Q}_1]) \wp (\texttt{num}\, N_1)$$

and

$$(\texttt{pc}\, L_4 + 2) \wp (\texttt{num}(N_4 + M_4 + 2)) \wp \mathcal{O}_2'' \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \mapsto_s \\ \mathcal{O}_1 \wp (\texttt{pc}\, 0) \wp (\mathcal{P}\, l_2\, E[\vec{Q}_2]) \wp (\texttt{num}\, N_2).$$

Since there is no dependence between the two instructions, I can permute the order in which the instructions are computed without effecting the observable state at the end of the execution of the two instructions.

The rest of the proof is obtained by inducting on the number of times the blocks Q_1 and Q_2 are computed. The other side of the transformation can be completed similarly. ■

I have provided a formal definition of observational equivalence for DLX programs which can be tackled using program transformations. As an example, I showed how to justify code-rescheduling from the definitions developed. This points out a subtle assumption in the code rescheduling, that the instructions in the two sequences are computed atomically – either one executes the entire sequence, or none of it. Without this assumption, code rescheduling cannot be justified in general. This opens up an interesting line of investigation into back end optimization using this meta-theoretic tool.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

My goal has been to analyze a meta-theory in which various issues regarding programming languages can be discussed. The first, and key requirement of a framework would be that it can specify the operational semantics of the programming language. However, I want the meta-theory to play a much more significant role than the specification of operational semantics alone. In particular, I want to use the meta-theory to study various interesting and challenging properties of programs. One key feature of a meta-theory should be to facilitate discussion of the programming language at various levels of detail — from high-level specifications down to abstract machines. The meta-theory should provide a uniform framework in which diverse properties — subject reduction, compiler optimizations, observational equivalence, and equivalence of different specifications — can be analyzed.

In this thesis I used FORUM as a meta-theory to study programming languages. FORUM provides a rich structure to proofs which was used to specify concurrency, higher-order functions, exceptions, state, and first-class continuations. I specified a fragment of $HO\pi$-calculus in FORUM to show how concurrent computations may be represented in FORUM. Next, I defined an untyped higher-order functional language, UML, which provides an exception mechanism and state and first-class continuations. UML, without first-class continuations, is untyped SML without data-types. UML encapsulates essential programming constructs which have been challenging to understand in more than one way. For example, modular specifications of UML have not been possible. The semantic analysis of the observational equivalence for Λ_{vs}, the functional core of UML augmented with state, has been very challenging. Next, I analyze the DLX architecture in FORUM. The executable specification of DLX architecture, to the best of my knowledge, has not been attempted. Formalizing low-level optimizations and observational equivalence for DLX programs has been very challenging.

Specifying UML modularly and declaratively has been challenging for formal systems because of the presence the various imperative features. I have provided modular and declarative specifications of the imperative features in UML. The claim that the specifications are modular is justified by the fact that I obtain the specification for UML by *literally putting together* my specifications for its different parts. The claim that the specifications are declarative is justified by the fact that my proofs regarding the evaluations in FORUM work by composing proofs using *cut rules* of FORUM.

As a result of my specification, evaluations become proofs in FORUM — formal objects which can be analyzed using the meta-theory of FORUM. I use this fact to study observational equivalence for Λ_{vs}. Using the proof structure of evaluations in FORUM, I have proved some of the challenging observational equivalences in the literature for Λ_{vs}-like languages. The nature of these proofs is very interesting. They seem to fall into two main categories. One kind of proofs essentially permute a given evaluation proof, typically using information regarding variable occurrences. The other kind of proofs are based on abstracting away details of function parameter from computations. The structure of proofs, richer logic, and *cut rules* play a key role in this analysis. However, the story is far from complete. Proofs of observational equivalence exhibit the need for a richer meta-theory for FORUM.

The declarative specification of the DLX pipeline, with its complex synchronizations, hazard resolution, call-forwarding, branch prediction and early-branch resolution, provides ample evidence of the flexibility of FORUM as a specification language for concurrent and imperative processes. The specifications allow me to prove the correctness of the pipeline with respect to the simple sequential evaluator for DLX programs. Further, I provide a definition of observational equivalence for DLX programs, and justify code rescheduling using the definitions. This effort highlights the key concept of blocks when discussing equivalence of DLX programs.

FORUM seems to provide an appropriate starting point as a meta-theory for present day programming languages. My results regarding the specification of $HO\pi$-calculus, UML and DLX prove that the proofs in FORUM are rich enough to represent a variety of computational paradigms. The analyses of the observational equivalence for Λ_{vs}, and DLX programs justifies the claim that FORUM can be used to study meta-theoretic properties also.

6.2 Future Work

The future work that I want to do in this area has three principal directions. First, I want to develop proof theory required to better specify imperative features and analyze proofs in FORUM. Second, I wish to study the derivation of abstract machines from high-level specifications in FORUM. Finally, I wish to consider other specification tasks. I describe each of these topics below.

Proof-theoretic challenges

Quantifiers for location names

Although FORUM is able to specify imperative features declaratively, there is one aspect which is not captured entirely by the specification. In the specification for Λ_{ve}, I used natural numbers to generate unique names for exceptions. The reason for using the sigctr was that I had to do inequality checks on the exception names when I searched the exception stack for an appropriate handler. The same problem would come up in the specification of

Λ_{vs} if the language permitted us to check for the equality of location names. The problem of the mismatch between restriction in $HO\pi$-calculus and universal quantifier is also related.

The usage of $\forall$ to represent creation of new location names is not entirely appropriate. This may be an overkill, because I instantiate the universal quantifier with location names only, not arbitrary values. In some sense, *I need a quantifier for pointers*. When the $\forall$ is introduced, it discharges a constant from the signature. Along with the discharging of the constant, it might be possible to manage inequality clauses between all the location names. In this sense, the new quantifier then may handle both the "newness" and "uniqueness" of the location names. A solution along these lines would allow for a completely logical specification of the exception mechanism. Further, a proper proof-theoretic understanding might help in the search for semantics for such languages.

Proof transformations

In chapter 4 we saw how proofs were manipulated to yield observational equivalences. There were two basic flavors to the proofs. First, the proof would essentially permute a given evaluation proof typically based on information regarding variable occurrences. Second, the proof would attempt to abstract the details of a function parameter from the computation of a program.

Some of the proofs were by induction on the height of proofs in FORUM. This might lead one to believe that there was not much uniformity to the transformations. Fortunately, just the contrary is true. For example, in lemma 4.23, I transform evaluations of (app M' Add_2) to evaluations of (app M' c) by induction on the height of the evaluation of (app M' Add_2). The shape of the evaluation of (app M' Add_2) will be as shown below. δ is the evaluation of (app Add_2 V).

$$\cfrac{\cfrac{\cfrac{\vdots}{\textit{Evaluation of}\,(\mathsf{app}\ \mathrm{Add}_2\ V)}}{\cfrac{\vdots}{\textit{Evaluation of}\,(\mathsf{app}\ \mathrm{Add}_2\ V)}}}{\cfrac{\vdots}{\textit{Sequent for start state of}\,(\mathsf{app}\ M'\ \mathrm{Add}_2)}}$$

The transformation replaces the proof fragment δ with the constant c. *The structure of the rest of the proof does not matter to the transformation.* The question is, how to study these transformations proof theoretically so that the above transformations can be specified compositionally instead of having to induct every time.

Deriving abstract machines from high-level specifications

One of the problems in language development is to show the link between the actual abstract machine that is implemented and the high-level semantics that one starts with. The traditional specification techniques are not able to present these different levels of abstractions. Hence, one has to mediate between dissimilar formalisms using some hairy induction arguments. I would like to logically transform the evaluator I have to a CEK style abstract machine for UML. A similar transformation was done for Λ_v like language in [HM92]. My work would extend it to the richer language UML. The transformation in [HM92] was not carried out entirely within the logic. I want to investigate whether the mixture of FORUM and continuation-passing-style specification will overcome some of the problem encountered in [HM92].

Specification of other aspects of programming languages

One of the natural questions that comes after having specified UML, is whether it can be typed in FORUM, *i.e.*, can I specify the static semantics of UML in FORUM? I have some preliminary ideas on this problem. The Subject Reduction theorem would then be a statement about the compositionality between the typing derivation and the evaluation.

The general setting of FORUM would also allow us to analyze other static information about the programs, such as *effects*. The proof theory may provide us insight into the logical nature of *effects*, if any.

Implementation of FORUM and interpreters for DLX

An implementation of a fragment of FORUM with first-order unification would suffice to play with interpreters for DLX pipelines. This could be a big step forward in understanding the role of FORUM as a prototyping language for such applications. The critical use of prototypes to experiment with new ideas and negotiate contracts is gaining much recognition recently. As such, this direction might result in some tangible applications of FORUM.

Appendix A

Proofs from chapter 3

In figure A.1 the constants from Σ_{ml} used for the translation of UML into FORUM is presented. In figure A.2 the translation from UML to FORUM is presented, in figure A.3 the translation for *Answers* to FORUM is presented, and in figure A.4 the translation from FORUM terms of type vl and tm to UML is presented. I prove lemmas A.1 and A.2 which implies lemmas 3.1, 3.4, 3.9 and 3.13. I use the notation from chapter 3 regarding freely in the appendix.

Lemma A.1 *Let M and N be UML terms, V and U be values in UML.*

1. $\phi(U[x := V]) \;=\; \phi(U)[x := \phi(V)]$.
2. $\mathcal{H}(M[x := V]) \;=\; \mathcal{H}(M)[x := \phi(V)]$.

Proof: The proof of the lemma works by mutual induction on the two claims. The induction is done on the structure of the term.

Proof for claim 1: The claim is vacuously proved in the cases where U is an exception name, or a constant in the language.

Case $U \;=\; z$, $z \not\equiv x$: LHS $= z =$ RHS

$$
\begin{array}{rcll}
\mathsf{abs} & : & (\mathsf{vl} \to \mathsf{tm}) \to \mathsf{vl} & \\
c & : & \mathsf{vl} & c \in \mathcal{B} \cup \mathcal{Z} \cup \{\bullet\} \\
\\
\langle . \rangle & : & \mathsf{vl} \to \mathsf{tm} & \\
f & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & f \in \mathcal{O} \\
\mathsf{app} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{cond} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{ifbr} & : & \mathsf{vl} \to \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{letval} & : & (\mathsf{vl} \to \mathsf{tm}) \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{letfun} & : & (\mathsf{vl} \to \mathsf{tm}) \to (\mathsf{vl} \to \mathsf{vl} \to \mathsf{tm}) \to \mathsf{tm} & \\
\mathsf{cell} & : & \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{read} & : & \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{write} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{ex} & : & \mathsf{ext} \to \mathsf{vl} & \\
\mathsf{exn} & : & (\mathsf{ext} \to \mathsf{tm}) \to \mathsf{tm} & \\
\mathsf{install} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{signal} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{catch} & : & \mathsf{tm} \to \mathsf{tm} & \\
\mathsf{jump} & : & \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} & \\
\\
\mathsf{get} & : & \mathsf{vl} \to (\mathsf{vl} \to o) \to o & \\
\mathsf{set} & : & \mathsf{vl} \to \mathsf{vl} \to (\mathsf{vl} \to o) \to o & \\
\mathsf{apply} & : & \mathsf{tm} \to \mathsf{tm} \to o & \\
\mathsf{uncaught} & : & \mathsf{ext} \to \mathsf{vl} \to o & \\
\mathsf{resume} & : & \mathsf{vl} \to \mathsf{vl} \to o & \\
\mathsf{cont} & : & \mathsf{vl} \to (\mathsf{vl} \to o) \to o & \\
\mathsf{eval} & : & \mathsf{tm} \to (\mathsf{vl} \to o) \to o &
\end{array}
$$

Figure A.1: Constants in Σ_{ml} used in translating UML to FORUM

$$
\begin{array}{rcll}
\phi(x) & = & x & \\
\phi(\lambda x.\, M) & = & \mathsf{abs}\ \lambda x : \mathsf{vl}.\ \mathcal{H}(M) & \\
\phi(c) & = & c & c \in \mathcal{Z} \cup \mathcal{B} \cup \{\bullet\} \\
\phi(l) & = & (\mathsf{ex}\ l) & l \in ExnNames
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{H}(V) & = & \langle \phi(V) \rangle \\
\mathcal{H}(f\ M\ N) & = & f\ \mathcal{H}(M)\ \mathcal{H}(N) \\
\mathcal{H}(M\ N) & = & \mathsf{app}\ \mathcal{H}(M)\ \mathcal{H}(N) \\
\mathcal{H}(\mathsf{if}\ M\ N\ P) & = & \mathsf{cond}\ \mathcal{H}(M)\ \mathcal{H}(N)\ \mathcal{H}(P) \\
\mathcal{H}(\mathsf{let\ val}\ x = M\ \mathsf{in}\ N) & = & \mathsf{letval}\ (\lambda x.\, \mathcal{H}(N))\ \mathcal{H}(M) \\
\mathcal{H}(\mathsf{let\ fun}\ f\, x = M\ \mathsf{in}\ N) & = & \mathsf{letfun}\ (\lambda f.\, \mathcal{H}(N))\ (\lambda f, x.\, \mathcal{H}(M)) \\
\mathcal{H}(\mathsf{ref}\ M) & = & \mathsf{cell}\ \mathcal{H}(M) \\
\mathcal{H}(\mathsf{deref}\ M) & = & \mathsf{read}\ \mathcal{H}(M) \\
\mathcal{H}(\mathsf{asg}\ M\ N) & = & \mathsf{write}\ \mathcal{H}(M)\ \mathcal{H}(N) \\
\mathcal{H}(\mathsf{exception}\ l\ M) & = & \mathsf{exn}\ \lambda l.\, \mathcal{H}(M) \\
\mathcal{H}(\mathsf{handle}\ M\ N\ P) & = & \mathsf{install}\ \mathcal{H}(M)\ \mathcal{H}(N)\ \mathcal{H}(P) \\
\mathcal{H}(\mathsf{raise}\ M\ N) & = & \mathsf{signal}\ \mathcal{H}(M)\ \mathcal{H}(N) \\
\mathcal{H}(\mathsf{callcc}\ M) & = & \mathsf{catch}\ \mathcal{H}(M) \\
\mathcal{H}(\mathsf{throw}\ M\ N) & = & \mathsf{jump}\ \mathcal{H}(M)\ \mathcal{H}(N)
\end{array}
$$

Figure A.2: Translating UML to FORUM

$$
\begin{array}{rcl}
\mathcal{A}(V, K) & = & (K, \phi(V)) \\
\mathcal{A}([\mathsf{pk}\ l, V\ ,]K) & = & (\mathsf{uncaught}\ l\ \phi_{ve}(V))
\end{array}
$$

Figure A.3: Translating answers in UML to FORUM

$$
\begin{array}{rcll}
\psi(x) & = & x & \\
\psi(\mathsf{abs}\ \lambda x.\, M) & = & \lambda x.\, \mathcal{L}(M) & \\
\psi(c) & = & c & c \in \mathcal{Z} \cup \mathcal{B} \cup \{\bullet\} \\
\psi(\mathsf{ex}\ l) & = & l & \\
 & & & \\
\mathcal{L}(\langle V \rangle) & = & \psi(V) & \\
\mathcal{L}(f\ M\ N) & = & f\ \mathcal{L}(M)\ \mathcal{L}(N) & f \in \mathcal{O} \\
\mathcal{L}(\mathsf{app}\ M\ N) & = & \mathcal{L}(M)\ \mathcal{L}(N) & \\
\mathcal{L}((\mathsf{ifbr}\ V\ N\ P)) & = & \mathsf{if}\ \psi(V)\ \mathcal{L}(N)\ \mathcal{L}(P) & b \in \mathcal{B} \\
\mathcal{L}(\mathsf{cond}\ M\ N\ P) & = & \mathsf{if}\ \mathcal{L}(M)\ \mathcal{L}(N)\ \mathcal{L}(P) & \\
\mathcal{L}(\mathsf{letval}\ R\ N) & = & \mathsf{let\ val}\ x = \mathcal{L}(N)\ \mathsf{in}\ \mathcal{L}(R\,x) & x\ \mathit{fresh} \\
\mathcal{L}(\mathsf{letfun}\ R_1\ R_2) & = & \mathsf{let\ fun}\ f\,x = \mathcal{L}(R_2\,f\,x)\ \mathsf{in}\ \mathcal{L}(R_1\,f) & f, x\ \mathit{fresh} \\
\mathcal{L}(\mathsf{cell}\ M) & = & \mathsf{ref}\ \mathcal{L}(M) & \\
\mathcal{L}(\mathsf{read}\ M) & = & \mathsf{deref}\ \mathcal{L}(M) & \\
\mathcal{L}(\mathsf{write}\ M\ N) & = & \mathsf{asg}\ \mathcal{L}(M)\ \mathcal{L}(N) & \\
\mathcal{L}(\mathsf{exn}\ R) & = & \mathsf{exception}\ l\ \mathcal{L}(R\,l) & l \in \mathit{ExnNames}, l\ \mathit{is\ fresh} \\
\mathcal{L}(\mathsf{install}\ M\ N\ P) & = & \mathsf{handle}\ \mathcal{L}(M)\ \mathcal{L}(N)\ \mathcal{L}(P) & \\
\mathcal{L}(\mathsf{signal}\ M\ N) & = & \mathsf{raise}\ \mathcal{L}(M)\ \mathcal{L}(N) & \\
\mathcal{L}(\mathsf{catch}\ M) & = & \mathsf{callcc}\ \mathcal{L}(M) & \\
\mathcal{L}(\mathsf{jump}\ M\ N) & = & \mathsf{throw}\ \mathcal{L}(M)\mathcal{L}(N) &
\end{array}
$$

Figure A.4: Translating FORUM terms to UML

Case $U = z$**,** $z \equiv x$ **:** LHS $= \phi(V) =$ RHS

Case $U = \lambda z.\, M \;\; z \not\equiv x$ **:**

If $z \equiv x$ then the equality is immediate. I write the case when $z \not\equiv x$.

LHS $= \phi((\lambda z.\, M)[x := V]) = \phi(\lambda z.\, (M[x := V])) = \mathsf{abs}\ \lambda z.\, \mathcal{H}(M[x := V])$

$= \mathsf{abs}\ \lambda z.\, (\mathcal{H}(M)[x := \phi(V)])$, by induction on claim 2.

$= (\mathsf{abs}\ \lambda z.\, \mathcal{H}(M))[x := \phi(V)] =$ RHS

This completes the proof of claim 1.

Proof for claim 2:

Case $M = U$**,** $U \in \mathit{Values}$ **:** By claim 1.

Case $M = (f\ N\ P)$**,** $f \in \mathcal{O}$ **:**

LHS $= f\, \mathcal{H}(N[x := V])\, \mathcal{H}(P[x := V])$, by induction on claim2

$= f\, \mathcal{H}(N)[x := \phi(V)]\, \mathcal{H}(P)[x := \phi(V)] =$ RHS

Case $M = (N\ P)$ **:**

LHS $= \mathsf{app}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])$, by induction on claim2

$= \mathsf{app}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)] =$ RHS

Case $M = (\mathsf{if}\ N\ P\ L)$ **:**

LHS $= \mathsf{cond}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])\ \mathcal{H}(L[x := V])$, by induction on claim2

$= \mathsf{cond}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)]\ \mathcal{H}(L)[x := \phi(V)] =$ RHS

Case $M = (\textsf{let val } z = N \textsf{ in } P)$ **:**

When $z \equiv x$ the result is immediate. I write the proof for $z \not\equiv x$.

LHS = $\textsf{letval}\ (\lambda z.\, \mathcal{H}(P[x := V]))\ \mathcal{H}(N[x := V])$, by induction on claim2

$= \textsf{letval}\ ((\lambda z.\, \mathcal{H}(P))[x := \phi(V)])\ \mathcal{H}(N)[x := V]$ = RHS

Case $M = (\textsf{let fun } f, z = N \textsf{ in } P)$ **:**

When $z \equiv x$ or $z \equiv f$ the result is immediate. I write the proof for $z \not\equiv x$ and $z \not\equiv f$.

LHS = $\textsf{letfun}\ (\lambda f.\, \mathcal{H}(P[x := V]))\ (\lambda f, z.\, \mathcal{H}(N[x := V]))$ by induction on claim2

$= \textsf{letfun}\ ((\lambda f.\, \mathcal{H}(P))[x := \phi(V)])\ ((\lambda f, z.\, \mathcal{H}(N))[x := \phi(V)])$ = RHS

Case $M = (\textsf{ref } N)$ **:**

LHS = $\textsf{cell}\ \mathcal{H}(N[x := V])$, by induction on claim2

$= \textsf{cell}\ \mathcal{H}(N)[x := \phi(V)]$ = RHS

Case $M = (\textsf{deref } N)$ **:**

LHS = $\textsf{read}\ \mathcal{H}(N[x := V])$, by induction on claim2

$= \textsf{read}\ \mathcal{H}(N)[x := \phi(V)]$ = RHS

Case $M = (\textsf{asg } N\ P)$ **:**

LHS = $\textsf{write}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])$, by induction on claim2

$= \textsf{write}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)]$ = RHS

Case $M = (\textsf{exception } l\ N)$ **:**

Note $x \not\equiv l$.

LHS = $\textsf{exn}\ l\mathcal{H}(N[x := V])$, by induction on claim2

$= \textsf{exn}\ l\mathcal{H}(N)[x := \phi(V)]$ = RHS

Case $M = (\textsf{raise}\ N\ P)$ **:**

LHS = $\textsf{raise}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])$, by induction on claim2

$= \textsf{raise}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)] =$ RHS

Case $M = (\textsf{handle}\ N\ P\ L)$ **:**

LHS = $\textsf{signal}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])\mathcal{H}(L[x := V])$, by induction on claim2

$= \textsf{signal}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)]\mathcal{H}(L)[x := \phi(V)] =$ RHS

Case $M = (\textsf{callcc}\ N)$ **:**

LHS = $\textsf{catch}\ \mathcal{H}(N[x := V])$, by induction on claim2

$= \textsf{catch}\ \mathcal{H}(N)[x := \phi(V)] =$ RHS

Case $M = (\textsf{throw}\ N\ P)$ **:**

LHS = $\textsf{jump}\ \mathcal{H}(N[x := V])\ \mathcal{H}(P[x := V])$, by induction on claim2

$= \textsf{jump}\ \mathcal{H}(N)[x := \phi(V)]\ \mathcal{H}(P)[x := \phi(V)] =$ RHS ■

The proof for lemma A.2 is a mutual induction exactly along the lines of lemma A.1. As the details do not reveal anything new I have not written down the proof.

Lemma A.2 *Let* $N : \textsf{tm}$*,* $U : \textsf{vl}$*, and* $V : \textsf{vl}$ *be FORUM terms.*

1. $\psi(U[x := V]) = \psi(U)[x := \psi(V)]$.

2. $\mathcal{L}(N[x := V]) = \mathcal{L}(N)[x := \psi(V)]$.

I now want to prove theorem 3.18. This theorem states that the natural semantics specification of Λ_{vse} and $\mathcal{E}_{ves}$, and the FORUM specification of Λ_{vse} are equivalent. I begin with the definition of a *configuration* and the translation of the configuration to FORUM. Configurations are the initial environments in which Λ_{vse} terms need to be evaluated. The

$$
\dfrac{\dfrac{\dfrac{}{\Sigma : \Phi, \mathsf{CL}_S\,;\, Q \wp \Delta_C \;\longrightarrow\; Q \wp \Delta_C}\,Id}{\vdots, \forall L, \Rightarrow L}}{\Sigma : \Phi, \mathsf{CL}_S\,;\, (C^o Q) \;\longrightarrow\; Q \wp \Delta}\;c
$$

$$
\dfrac{\vdots, \text{Associativity and commutativity of } \wp}{\Sigma : \Phi, \mathsf{CL}_S\,;\, (C^o Q) \;\longrightarrow\; Q \wp \Gamma}
$$

Figure A.5: C is a configuration

translation of a configuration to FORUM returns a λ which places its arguments in an environment in which all the cells and exceptions have been declared, and approporaitely quantified.

Definition A.3 [Configurations in Λ_{vse}] A conifguration, C is a pair of state, S, and set of exception names, Ex, such that forall $l \in \mathsf{dom}(S)$, $FV(S(l)) \subset \mathsf{dom}(S) \cup Ex$. ■

Definition A.4 [Translating Configurations to FORUM] $C =_{def} \langle S, Ex \rangle$, be a configuration. The translation of C into FORUM is a term of type $o \rightarrow o$, written as C^o.

$$
\begin{aligned}
&C^o \;=_{def} \\
&\lambda u : o. \forall\, P_S, l_S.\ \mathsf{getC}(P_1, l_1) \Rightarrow \mathsf{setC}(P_1, l_1) \Rightarrow \ldots \Rightarrow \mathsf{getC}(P_n, l_n) \Rightarrow \mathsf{setC}(P_n, l_n) \Rightarrow \\
&\qquad\qquad [u \wp \Gamma_S \wp (\mathsf{sigctr}\ l_{Ex}) \wp (\mathsf{exnst\ nil})]
\end{aligned}
$$

■

The domain of a configuration, $C =_{def} \langle S, Ex \rangle$, written as $\mathsf{dom}(C)$, is the union of $\mathsf{dom}(S)$ and Ex. In figure A.5 I show a proof figure, *Compl*, which I will use to end the computations in Λ_{vse}. Let $C = \langle S, Ex \rangle$.

$$
\begin{aligned}
C^o Q \;=\; &\forall\, P_S, l_S.\ \mathsf{getC}(P_1, l_1) \Rightarrow \mathsf{setC}(P_1, l_1) \Rightarrow \ldots \Rightarrow \mathsf{getC}(P_n, l_n) \Rightarrow \mathsf{setC}(P_n, l_n) \Rightarrow \\
&\qquad [Q \wp \Gamma_S \wp (\mathsf{sigctr}\ l_{Ex}) \wp (\mathsf{exnst\ nil})]
\end{aligned}
$$

$l_C =_{def} l_{Ex}$, $\Gamma_C =_{def} \Gamma_S$, $\Delta_C =_{def} \Gamma_C \wp(\mathsf{sigctr}\ l_C) \wp(\mathsf{exnst\ nil})$, and Δ is some permutation of Δ_C. I restate Theorem 3.18.

Theorem A.5 (Correspondence theorem for Λ_{vse}) *Let $M \in \Lambda_{vse}$, $A \in Answers_{vse}$, C_2 be a configuration and C_1 be a configuration such that $FV(M) \subset \mathsf{dom}(C)$.*

$\langle M, C_1 \rangle \Downarrow \langle A, C_2 \rangle$ *if and only if* $\mathsf{eval}(\mathcal{H}(M), C_1, \mathcal{A}(A, K), C_2)$

Proof: Assume $M \in \Lambda_{vse}$, $A \in Answers_{vse}$, C_2 is a configuration and C_1 is a configuration such that $FV(M) \subset \mathsf{dom}(C)$.

Left to right direction : Proof is by induction on the height of the evaluation tree in the natural semantics. I do a case analysis on the structure of M. There are in total thirty-nine rules in natural semantics that I have to analyze. Since many of the cases are repetetive, I will prove the ones which have different features and leave out the proofs of the rest. In the proofs I show, I will not show parts of the signature and intuitionistic context. For example, I do not write Σ_{ves} in the signature, and $\mathcal{E}_{ves}$ in the intuitionistic context, since they are present in all the exhibited sequents.

Case $M = V$: There is only one possible evaluation tree. The proof in FORUM follows trivially from *Identity*.

Case $M = (N\ P)$: There are four possible evaluation trees. Suppose the evaluation is

$$\frac{\langle N, C_1 \rangle \Downarrow \langle \lambda x.\, Q, C_2 \rangle \quad \langle P, C_2 \rangle \Downarrow \langle U, C_3 \rangle \quad \langle Q[x := U], C_3 \rangle \Downarrow \langle V, C_4 \rangle}{\langle (N\ P), C_1 \rangle \Downarrow \langle V, C_4 \rangle}$$

The proof is essentially the one given in section 3.1 after theorem 3.3. There are three more natural semantics trees, each for the fact that we have an uncaught exception. The proof for these cases is completed as shown in section 3.3 after theorem 3.12.

Case $M = (f\ N\ P)$, $f \in \mathcal{O}$ **:** In this case, there are three possible natural semantics which are applicable. The proof is along the lines of the case $M = (N\ P)$.

Case $M = (\mathsf{if}\ N\ P\ Q)$ **:** In this case, there are six possible natural semantics which are applicable. I show one of the cases, the others may be completed similary. Suppose the last natural semantics rule used is :

$$\frac{\langle N, C_1\rangle \Downarrow \langle \mathsf{true}, C_2\rangle \quad \langle P, C_2\rangle \Downarrow \langle V, C_3\rangle}{\langle(\mathsf{if}\ N\ P\ Q), C_1\rangle \Downarrow \langle V, C_3\rangle}$$

The evaluation trees for $\langle N, C_1\rangle \Downarrow \langle \mathsf{true}, C_2\rangle$ and $\langle P, C_2\rangle \Downarrow \langle V, C_3\rangle$ are smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$, $P_1 =_{def} \mathcal{H}(P)$, $Q_1 =_{def} \mathcal{H}(Q)$, and $V_1 =_{def} \phi(V)$. Thus, by using induciton hypothesis I get proofs δ_1, and δ_2 of

- $K_1 : \mathcal{E}_{ves}\,;\, C_2^o(K_1\,\mathsf{true}) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ K_1)$, and
- $K_2 : \mathcal{E}_{ves}\,;\, C_3^o(K_2\,V_1) \longrightarrow C_2^o(\mathsf{eval}\ P_1\ K_2)$, respectively.

The required proof is constructed below. In the proof,
$L_1 =_{def} \lambda v.\,(\mathsf{eval}\ ((\mathsf{ifbr}\ v\ P_1\ Q_1))\ K)$, and $(L_1\,\mathsf{true}) =_\beta (\mathsf{eval}\ ((\mathsf{ifbr}\ \mathsf{true}\ P_1\ Q_1))\ K)$.

$$\cfrac{\cfrac{\cfrac{\cfrac{C_3^o(K\,V_1) \overset{\sigma_2}{\longrightarrow} C_2^o(\mathsf{eval}\ P_1\ K)}{C_3^o(K\,V_1) \longrightarrow (\mathsf{eval}\ ((\mathsf{ifbr}\ \mathsf{true}\ P_1\ Q_1))\ K)\wp\Delta_{C_2}}}{\vdots, \forall R, \Rightarrow R}}{C_3^o(K\,V_1) \longrightarrow C_2^o(L_1\,\mathsf{true})} \qquad \overset{\sigma_1}{C_2^o(L_1\,\mathsf{true}) \longrightarrow (\mathsf{eval}\ N_1\ L_1)\wp\Delta_{C_1}}}{\cfrac{\Sigma_{C_1}, K : \mathsf{CL}_{S_1}\,;\, C_3^o(K\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ L_1)\wp\Delta_{C_1}}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1}\,;\, C_3^o(K\,V_1) \longrightarrow (\mathsf{eval}\ (\mathsf{if}\ N_1\ P_1\ Q_1)\ K)\wp\Delta_{C_1}}}$$

In the above proof, σ_1 is obtained by δ_1 by a *CutS* rule K_1 with L_1, similarly σ_2 is obtained from δ_2 by a *CutS* rule on K_2 with K.

Suppose the last natural semantics rule used is :

$$\frac{\langle N, C_1\rangle \Downarrow \langle [\mathsf{pk}\ l\ V], C_2\rangle}{\langle (\mathsf{if}\ N\ P\ Q), C_1\rangle \Downarrow \langle V, C_2\rangle}$$

The evaluation tree for $\langle N, C_1\rangle \Downarrow \langle [\mathsf{pk}\ l\ V], C_2\rangle$ is smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$, $P_1 =_{def} \mathcal{H}(P)$, $Q_1 =_{def} \mathcal{H}(Q)$, and $V_1 =_{def} \phi(V)$. Thus, by using induciton hypothesis I get proof δ_1 of

- $K_1 : \mathcal{E}_{ves}\, ; C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ K_1)$.

The required proof is constructed below.

$$\frac{\overset{\sigma_1}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1}\, ; C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow (\mathsf{eval}\ N_1\ L_1) \wp \Delta_{C_1}}}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1}\, ; C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow (\mathsf{eval}\ (\mathsf{if}\ N_1\ P_1\ Q_1)\ K) \wp \Delta_{C_1}}\ \textit{backchain}$$

Where σ_1 is obtained from δ_1 using *CutS*. The proof, for the other natural semantics rules for if are completed in a similar manner.

Case $M = (\mathsf{let\ val}\ x = N\ \mathsf{in}\ P)$ **:** This case is handled exactly like $(\lambda x.\, P)\ N$.

Case $M = (\mathsf{let\ fun}\ f, x = N\ \mathsf{in}\ P)$ **:** The last evaluation rule for letfun will be :

$$\frac{\langle P[f := \lambda x.\, \mathsf{let\ fun}\ f\ x = N\ \mathsf{in}\ N], C_1\rangle \Downarrow \langle A, C_2\rangle}{\langle \mathsf{let\ fun}\ f\ x = N\ \mathsf{in}\ P, C_1\rangle \Downarrow \langle A, C_2\rangle}$$

The evaluation tree for $\langle P[f := \lambda x.\, \mathsf{let\ fun}\ f\ x = N\ \mathsf{in}\ N], C_1\rangle \Downarrow \langle A, C_2\rangle$ is smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$, $P_1 =_{def} \mathcal{H}(P)$, and $V_1 =_{def} \phi(V)$. Thus, by using induciton hypothesis I get the proof δ_1 of

- $K_1 : \mathcal{E}_{ves}\, ; C_2^o\mathcal{A}(A, K_1) \longrightarrow C_1^o(\mathsf{eval}\ P_1[f := (\mathsf{abs}\ \lambda x.\, \mathsf{let\ fun}\ f\ x = N_1\ \mathsf{in}\ N_1)]\ K_1)$.

The required proof is constructed below. Let $Q =_{def} (\mathsf{abs}\ \lambda x.\, \mathsf{let\ fun}\ f\ x = N_1\ \mathsf{in}\ N_1)$.

$$\cfrac{\overset{\sigma_1}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1} ; C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow (\mathsf{eval}\ P_1[f := Q]\ L_1) \wp \Delta_{C_1}}}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1} ; C_2^o\mathcal{A}(A, K) \longrightarrow (\mathsf{eval}\ \mathcal{H}(\mathsf{let\ fun}\ f\ x = N\ \mathsf{in}\ P)\ K) \wp \Delta_{C_1}}\ \mathit{backchain}$$

Where σ_1 is obtained from δ_1 using *CutS*.

Case $M = (\mathsf{ref}\ N)$ **:** I consider the case when the last evaluation rule for **ref** is :

$$\frac{\langle N, C_1\rangle \Downarrow \langle V, C_2\rangle}{\langle \mathsf{ref}\ N, C_1\rangle \Downarrow \langle l, C_2[l \mapsto V]\rangle}\ l \notin \text{State in } C_2$$

The evaluation tree for $\langle N, C_1\rangle \Downarrow \langle V, C_2\rangle$ is smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$, and $V_1 =_{def} \phi(V)$. Thus by using induction hypothesis I get proof δ_1 of

- $K_1 : \mathcal{E}_{ves} ; C_2^o(K\,V) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ K_1)$.

The required proof is constructed below. Let $C_3 =_{def} \langle S_2[l \mapsto V], Ex_2\rangle$, where $C_2 =_{def} \langle S_2, Ex_2\rangle$, and $L_1 =_{def} \lambda v. \forall P, l. \mathsf{getC}(P, l) \Rightarrow \mathsf{setC}(P, l) \Rightarrow [(K\,l) \wp (P\,v)]$.

$$\cfrac{\cfrac{\overset{\mathit{Compl}}{\Sigma_{C_3}, P, l : \mathsf{CL}_{S_3} ; C_3^o(K\,l) \longrightarrow (K\,l) \wp (P\,V_1) \wp \Delta_{C_2}}}{\cfrac{\vdots, \forall R, \Rightarrow R}{C_3^o(K\,l) \longrightarrow C_2^o(L_1\,V_1)}} \qquad \overset{\sigma_1}{C_2^o(L_1\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ L_1) \wp \Delta_{C_1}}}{\cfrac{\Sigma_{C_1}, K : \mathsf{CL}_{S_1} ; C_3^o(K\,l) \longrightarrow (\mathsf{eval}\ N_1\ L_1) \wp \Delta_{C_1}}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1} ; C_3^o(K\,l) \longrightarrow (\mathsf{eval}\ (\mathsf{cell}\ N_1)\ K) \wp \Delta_{C_1}}\ \mathit{backchain}}$$

Where σ_1 is obtained from δ_1 using *CutS*. The other case when the evaluation of N returns an uncaught exception is treated similarly.

Case $M = (\mathsf{deref}\ N)$ **:** I consider the case when the last evaluation rule for **deref** is :

$$\frac{\langle N, C_1\rangle \Downarrow \langle [\mathsf{pk}\ l\ V], C_2\rangle}{\langle \mathsf{deref}\ N, C_1\rangle \Downarrow \langle [\mathsf{pk}\ l\ V], C_2\rangle}$$

The evaluation tree for $\langle N, C_1\rangle \Downarrow \langle[\mathsf{pk}\ l\ V], C_2\rangle$ is smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$, and $V_1 =_{def} \phi(V)$. Thus, by using induction hypothesis I get proof δ_1 of

- $K_1 : \mathcal{E}_{ves}\ ;\ C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ K_1)$.

The required proof is constructed below. Let $L_1 =_{def} \lambda v.(\mathsf{get}\ v\ K)$.

$$\cfrac{\begin{array}{c}\sigma_1\\ \Sigma_{C_1}, K : \mathsf{CL}_{S_1}\ ;\ C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow (\mathsf{eval}\ N_1\ L_1) \wp \Delta_{C_1}\end{array}}{\Sigma_{C_1}, K : \mathsf{CL}_{S_1}\ ;\ C_2^o(\mathsf{uncaught}\ l\ V_1) \longrightarrow (\mathsf{eval}\ (\mathsf{read}\ N_1)\ K) \wp \Delta_{C_1}}\ \mathit{backchain}$$

Where σ_1 is obtained from δ_1 using *CutS*. The other case when evaluation of N returns a value is treated similarly. Essentially, the computation continues with $(\mathsf{get}\ l\ K)$, where l would be the value returned by N.

Case $M = (\mathsf{asg}\ N\ P)$ **:** There are three natural semantics rules for asg. The proofs for all of these is completed along the lines of the proof after theorem 3.8.

Case $M = (\mathsf{exception}\ l\ N)$ **:** This case is rather straightforward given the examples above. It is handled along the lines of ref.

Case $M = (\mathsf{raise}\ N\ P)$ **:** This case is rather straightforward given the examples above. It is handled along the lines of app.

Case $M = (\mathsf{handle}\ N\ P\ Q)$ **:** Suppose the last rule in the evalution was

$$\frac{\langle P, C_1\rangle \Downarrow \langle l, C_2\rangle \quad \langle Q, C_2\rangle \Downarrow \langle W, C_3\rangle \quad \langle N, C_3\rangle \Downarrow \langle[\mathsf{pk}\ l\ U], C_4\rangle \quad \langle (W\ U), C_4\rangle \Downarrow \langle V, C_5\rangle}{\langle \mathsf{handle}\ N\ P\ Q, C_1\rangle \Downarrow \langle V, C_5\rangle}$$

The evaluation trees for $\langle P, C_1\rangle \Downarrow \langle l, C_2\rangle$, $\langle Q, C_2\rangle \Downarrow \langle W, C_3\rangle$, $\langle N, C_3\rangle \Downarrow \langle[\mathsf{pk}\ l\ U], C_4\rangle$, and $\langle (W\ U), C_4\rangle \Downarrow \langle V, C_5\rangle$ are smaller than the evaluation of M. Let $N_1 =_{def} \mathcal{H}(N)$,

$P_1 =_{def} \mathcal{H}(P)$, $Q_1 =_{def} \mathcal{H}(Q)$, $V_1 =_{def} \phi(V)$, $U_1 =_{def} \phi(U)$, and $W_1 =_{def} \phi(W)$. Thus, by using induciton hypothesis I get proofs δ_1, δ_2, δ_3, and δ_4 of

- $K_1 : \mathcal{E}_{ves} \,;\, C_2^o(K_1\,(\mathsf{ex}\ l)) \longrightarrow C_1^o(\mathsf{eval}\ P_1\ K_1)$,
- $K_2 : \mathcal{E}_{ves} \,;\, C_3^o(K_2\,W_1) \longrightarrow C_2^o(\mathsf{eval}\ Q_1\ K_2)$,
- $K_3 : \mathcal{E}_{ves} \,;\, C_4^o(\mathsf{uncaught}\ l\ U_1) \longrightarrow C_3^o(\mathsf{eval}\ N_1\ K_3)$, and
- $K_4 : \mathcal{E}_{ves} \,;\, C_5^o(K_4\,V) \longrightarrow C_4^o(\mathsf{eval}\ (\mathsf{app}\ W_1\ U_1)\ K_4)$, respectively.

The required proof is constructed below.
Let $L_1 =_{def} \lambda v.\,(\mathsf{isexn}\ v\ \lambda w.\,(\mathsf{eval}\ Q_1\ \lambda u.\,(\mathsf{push}\ w\ u\ K\ N_1)))$,
$L_2 =_{def} \lambda w.\,(\mathsf{eval}\ Q_1\ \lambda u.\,(\mathsf{push}\ w\ u\ K\ N_1))$, and $L_3 =_{def} \lambda u.\,(\mathsf{push}\ l\ u\ K\ N_1)$.
Note $(L_1\,(\mathsf{ex}\ l)) =_{def} (\mathsf{isexn}\ (\mathsf{ex}\ l)\ \lambda w.\,(\mathsf{eval}\ Q_1\ \lambda u.\,(\mathsf{push}\ w\ u\ K\ N_1)))$,
$(L_2\,l) =_{def} (\mathsf{eval}\ Q_1\ \lambda u.\,(\mathsf{push}\ l\ u\ K\ N_1))$, and $(L_3\,W_1) =_{def} (\mathsf{push}\ l\ W_1\ K\ N_1)$

$$\dfrac{\dfrac{\dfrac{\overset{\gamma_1}{\Sigma_{C_2} : \mathsf{CL}_{C_2} \,;\, C_5^o(K\,V_1) \longrightarrow (L_1\,(\mathsf{ex}\ l))\wp\Delta_{C_2}}}{\vdots\,\forall R, \Rightarrow R}}{C_5^o(K\,V_1) \longrightarrow C_2^o(L_1\,(\mathsf{ex}\ l))} \qquad \overset{\sigma_1}{C_2^o(L_1\,(\mathsf{ex}\ l)) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ L_1)}}{\dfrac{\Sigma_{C_1} : \mathsf{CL}_{C_1} \,;\, C_2^o(K\,V_1) \longrightarrow C_1^o(\mathsf{eval}\ N_1\ L_1)}{\Sigma_{C_1} : \mathsf{CL}_{C_1} \,;\, C_5^o(K\,V_1) \longrightarrow C_1^o(\mathsf{eval}\ (\mathsf{install}\ N_1\ P_1\ Q_1)\ K)}}$$

To complete the proof I am left with constructing γ_1.

$$\dfrac{\dfrac{\dfrac{\overset{\gamma_2}{\Sigma_{C_3} : \mathsf{CL}_{C_3} \,;\, C_5^o(K\,V_1) \longrightarrow (L_2\,l)\wp\Delta_{C_3}}}{\vdots\,\forall R, \Rightarrow R}}{\Sigma_{C_2} : \mathsf{CL}_{C_2} \,;\, C_5^o(K\,V_1) \longrightarrow C_3^o(L_2\,l)} \qquad \dfrac{\dfrac{}{\Sigma_{C_2} : \mathsf{CL}_{C_2} \,;\, C_2^o(L_2\,l) \longrightarrow (L_2\,l)\wp\Delta_{C_2}}}{\Sigma_{C_2} : \mathsf{CL}_{C_2} \,;\, C_2^o(L_2\,l) \longrightarrow (L_1\,(\mathsf{ex}\ l))\wp\Delta_{C_2}}}{\Sigma_{C_2} : \mathsf{CL}_{C_2} \,;\, C_5^o(K\,V_1) \longrightarrow (L_1\,(\mathsf{ex}\ l))\wp\Delta_{C_2}}$$

γ_2 is constructed below.

$$\dfrac{\dfrac{\dfrac{\gamma_3}{C_5^o(K\,V_1) \longrightarrow (\mathsf{push}\ l\ W_1\ K\ N_1) \wp \Delta_{C_4}}}{\vdots\ \forall R, \Rightarrow R} }{C_5^o(K\,V_1) \longrightarrow C_4^o(L_3\,W_1)} \qquad \dfrac{\sigma_2}{C_4^o(L_3\,W_1) \longrightarrow (\mathsf{eval}\ Q_1\ L_3) \wp \Delta_{C_3}}$$

$$\overline{C_5^o(K\,V_1) \longrightarrow (\mathsf{eval}\ Q_1\ L_3) \wp \Delta_{C_3}}$$

γ_3 is constructed below.

$$\dfrac{\dfrac{\sigma_3}{C_5^o(K\,V_1) \longrightarrow (\mathsf{eval}\ N_1\ \lambda v.\,(\mathsf{pop}\ (K\ v))) \wp \Gamma_{C_4} \wp (\mathsf{sigctr}\ l_{C_4}) \wp (\mathsf{exnst}\ (\mathsf{pkt}\ l\ W_1\ K) :: \mathsf{nil})}}{C_5^o(K\,V_1) \longrightarrow (\mathsf{push}\ l\ W_1\ K\ N_1) \wp \Gamma_{C_4} \wp (\mathsf{sigctr}\ l_{C_4}) \wp (\mathsf{exnst}\ \mathsf{nil})}$$

The proof σ_i is constructed from δ_i, $i \in [1,2]$. σ_3 is obtained from δ_3. δ_3 results in ($\mathsf{uncaught}\ l\ U_1$), under the assumption that the initial exception stack is empty. However, the excetion stack is ($\mathsf{exnst}\ (\mathsf{pkt}\ l\ W_1\ K) :: \mathsf{nil}$). I change δ_3 to reflect the intial exception stack, which will catch the exception and evaluate ($\mathsf{apply}\ W_1\ U_1\ K$). The evaluation of ($\mathsf{apply}\ W_1\ U_1\ K$) is completed using δ_4. The other cases for handle are handled similarly.

Right to left direction : In this direction, I induct on height of the sequent proof in FORUM. I then do a case analyses based on the outermost term constructor for the term being evaluate. The proof in this direction is very similar in nature to the one for the other direction. I illustrate the basic strategy using ($\mathsf{asg}\ N\ P$) as an example, and do not write down the other cases. The computation for ($\mathsf{asg}\ N\ P$) in FORUM is as shown below. $P_1 =_{def} \mathcal{H}(P)$, $N_1 =_{def} \mathcal{H}(N)$, $A \in Answers_{vse}$, $V_1 =_{def} \phi(V)$, $L_1 =_{def} \lambda v.\,(\mathsf{eval}\ P_1\ \lambda u.\,(\mathsf{set}\ v\ u\ K))$, and $L_2 =_{def} \lambda u.\,(\mathsf{set}\ l\ u\ K)$.

$$\dfrac{\gamma_1 \qquad \dfrac{\dfrac{\vdots\ \mathit{Computation\ of}\, N_1,\ \mathrm{Pt1}}{C_4^o\mathcal{A}(A,K) \longrightarrow (\mathsf{eval}\ N_1\ L_1) \wp \Delta_{C_1}}}{}}{\Sigma_{C_1} : \mathsf{CL}_{C_1}\,;\, C_4^o\mathcal{A}(A,K) \longrightarrow (\mathsf{eval}\ (\mathsf{asg}\ N_1\ P_1)\ K) \wp \Delta_{C_1}}$$

$$\dfrac{\dfrac{\vdots\ \mathit{Computation\ of}\, P_1,\ \mathrm{Pt1}}{C_4^o\mathcal{A}(A,K) \longrightarrow (\mathsf{eval}\ P_1\ L_2) \wp \Delta_{C_2}}}{C_4^o\mathcal{A}(A,K) \longrightarrow (\mathsf{eval}\ l\ L_1) \wp \Delta_{C_2}}$$

Where γ_1 is shown below.

$$\dfrac{\dfrac{\dfrac{\textit{Compl}}{C_4^o\mathcal{A}(A,K)\ \longrightarrow\ (K\ \bullet)\wp\,\Gamma\,\wp\,(P\,V_1)\,\wp\,(\mathsf{sigctr}\ l_{C_3})\,\wp\,(\mathsf{exnst\ nil})}}{C_4^o\mathcal{A}(A,K)\ \longrightarrow\ (\mathsf{set}\ l\ V_1\ K)\,\wp\,\Gamma\,\wp\,(P\,U_1)\,\wp\,(\mathsf{sigctr}\ l_{C_3})\,\wp\,(\mathsf{exnst\ nil})}}{C_4^o\mathcal{A}(A,K)\ \longrightarrow\ (\mathsf{eval}\ P_1\ L_2)\,\wp\,\Delta_{C_2}}$$

From this proof the computations for $\langle N, C_1\rangle \Downarrow \langle l, C_2\rangle$ and $\langle P, C_2\rangle \Downarrow \langle l, C_3\rangle$ can be extracted easily. Next, the variable conditions in the proof imply that the state can be updated suitable so that the final answer is $\langle \bullet, C_3[l \mapsto V]\rangle$. However, if the evaluation of N_1 raised an uncaught exception, ($\mathsf{uncaught}\ l\ V_1$), then the proof would have ended with the *Compl* construction at the point Pt1. In this case, I can get a computation of $\langle N, C_1\rangle \Downarrow \langle[\mathsf{pk}\ l\ V], C_2\rangle$. The case when N_1 evaluates to a value, and P_1 raises an uncaught exception, ($\mathsf{uncaught}\ l\ V_1$), is handled similarly.

■

Bibliography

[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[Abr90] S. Abramsky. The lazy λ-calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[AC87] T. Agarwal and J. Cocke. High performance reduced instruction set processors. Technical report, IBM, 1987.

[AJ89] Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *16^{th} Symp. Principles of Programming Languages*, pages 293–302. ACM, 1989.

[AL93] M. Aagaard and M. Leeser. A framework for specifying and designing pipelines. In *IEEE International Conference on Computer Design*, pages 548–551, 1993.

[AO89] S. Abramsky and L. Ong. Full abstraction in the lazy λ-calculus. Technical report, Imperial College, 1989. To appear in Information and Computation.

[AP90] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proceeding of the Seventh International Conference on Logic Programming, Jerusalem*, May 1990.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[AST89] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. The IBM 360 model 91: Machine philosophy and instruction handling. In *IBM J. of Research and Developement*, pages 8–24, 1989.

[Bar84] Hank Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, revised edition, 1984.

[Blo59] E. Bloch. The engineering design of the stretch computer. In *Proc. Fall Joint Computer Conference*, pages 48–59, 1959.

[BR90] Egon Börger and Dean Rosenzweig. From prolog algebras towards wam – a mathematical study of implementations. In *Computer Science Logic*, volume 533, pages 31–66. Springer-Verlag, 1990.

[Buc62] W. Bucholtz. *Planning a computer system : Project Stretch.* McGraw Hill, 1962.

[CF94] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *Symposium on Theoretical Aspects of Computer Software*, 1994.

[Chi94] Jawahar Chirimar. What can continuations observe in the lazy λ-calculus? Technical report, University of Pennsylvania, April 1994.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[DH89] R.K. Dybvig and R. Heib. Engines from continuations. In *Computing Languages*, volume 14(2), pages 109–123, 1989.

[FF86] M. Felleisen and D.P. Friedman. Control operators, the SECD machine and the λ-calculus. In M. Wirsing, editor, *Formal Descriptions of Programming Concepts-III*, pages 193–217. North-Holland, 1986.

[FH92] M. Felleisen and R. Heib. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[Fri88] D.P. Friedman. Applications of continuations. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1988.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir92] Jean-Yves Girard. A fixpoint theorem in linear logic. A message posted on the linear@cs.stanford.edu mailing listing, February 1992.

[Gor79] M. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[Han90] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, August 1990.

[HM91] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32–42, Amsterdam, July 1991.

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992. Invited to a special issue of papers selected from the 1990 Lisp and Functional Programming Conference.

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The Definition of Standard ML: Version 3. Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh, May 1989.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, October 1969.

[HP90] J. Hennesy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufman Publishers, Inc., 1990.

[HSH90] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. 1. Clauses as rules. *Journal of Logic and Computation*, pages 261–283, December 1990.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of STACS 1987*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, March 1987.

[Kle64] Stephen Cole Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1964.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mil90] Robin Milner. Functions as processes. Research Report 1154, INRIA, 1990.

[Mil93] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in Lecture Notes in Computer Science, pages 242–265. Springer-Verlag, 1993.

[Mil94] Dale Miller. A multiple-conclusion meta-logic: Extended abstract. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, Paris, July 1994. To Appear.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mor68] J.H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, Massachusets Institute of Technology, 1968.

[MP92] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its metatheory in Elf. In Lars Hallnäs, editor, *Extensions of Logic Programming.* Springer-Verlag LNCS, 1992. To appear. A preliminary version is available as Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, September 1992.

[MS88] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proc. 15^{th} Annual ACM Symp. on Principles of Programming Languages*, pages 191–203, San Diego, 1988.

[MT92] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In A. Scedrov, editor, *Proceedings of LICS'92*, pages 186–197, 1992.

[OT92] Peter W. O'Hearn and Robert D. Tennent. Semantics of local variables. Technical Report ECS-LFCS-92-192, Laboratory for the Foundations of Computer Science, University of Edinburgh, January 1992.

[OT93] Peter W. O'Hearn and Robert D. Tennent. Relational parametricity and local variables. In *Proc. 20^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 171–184, 1993.

[Plo76] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[PS93] A.M. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages (Technical Report YALEU/DCS/RR-968, Yale University)*, 1993.

[RC86] Jonathan Ress and William Clinger. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21:37–79, 1986.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[Rey72] J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972.

[Rey81a] J. C. Reynolds. *The Craft of Programming.* Series in Computer Science. Prentice-Hall, 1981.

[Rey81b] J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[San92a] D. Sangiorgi. *Expressing Mobility in Process Algebras : First Order and Higher Order Paradigms.* PhD thesis, University of Edinburgh, 1992.

[San92b] D. Sangiorgi. The lazy λ calculus in a concurrency scenario. In *7th LICS Conf.* IEEE Computer Society Press, 1992.

[Set89] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley Pub. Co., 1989.

[SF92] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 288–298, 1992.

[SH93] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE, June 1993.

[Sie93] Kurt Sieber. New steps towards full abstraction for local variables. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages (Technical Report YALEU/DCS/RR-968, Yale University)*, pages 88–100, Copenhagen, Denmark, 1993.

[Ste78] Guy L. Steele. Rabbit: A compiler for Scheme. Technical report, MIT Artificial Intelligence Laboratory, 1978.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[SW74] C. Strachey and C.P. Wadsworth. Continuations: A mathematical semantics for handling full jump. Technical Report PRG-11, Oxford university Computing Laboratory, 1974.

[Tho70] J.E. Thornton. *Design of a computer, the Control Data 6600.* Foresman, 1970.

[TK93] S. Tahar and R. Kumar. A formalization of a hierarchical model for risc processors. In *Proceedings of EuroARCH'93.* Springer Verlag, 1993.

[Wan80] M. Wand. Continuation based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[WF91] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Rice University, 1991.