Provably Correct Compiler Generation

Jens Palsberg

Ph.D. thesis

Computer Science Department
Aarhus University

October 1992

Abstract

We have designed, implemented, and proved the correctness of a compiler generator that accepts action semantic descriptions of imperative programming languages. We have used it to generate compilers for both a toy language and a non-trivial subset of Ada. The generated compilers emit absolute code for an abstract RISC machine language that is assembled into code for the SPARC and the HP Precision Architecture. The generated code is an order of magnitude better than that produced by compilers generated by the classical systems of Mosses, Paulson, and Wand. Our machine language needs no runtime type-checking and is thus more realistic than those considered in previous compiler proofs. We use solely algebraic specifications; proofs are given in the initial model. The use of action semantics makes the processable language specifications easy to read and pleasant to work with. We view our compiler generator as a promising first step towards user-friendly and automatic generation of realistic and provably correct compilers.

Ackowledgements

The author thanks Peter Mosses for advice and support. The author also thanks Peter Ørbæk for implementing the Cantor system.

Overviews of this thesis has been presented at ESOP'92, European Symposium on Programming [63] and ICCL'92, Fourth IEEE International Conference on Computer Languages [62].

Advisory committee:	Peter Mosses
	Ole Lehrmann Madsen
	Mogens Nielsen
External examiner:	Neil Jones

Resumé

Vi har designet, implementeret og bevist korrektheden af en oversætter generator som tager aktions-semantiske beskrivelser af imperative programmeringssprog som input. Vi har brugt den til at generere oversættere for både et legetøjs-sprog og en ikke-triviel delmængde af Ada. De genererede oversættere producerer absolut kode for en abstrakt RISC maskine som derefter oversættes til kode til SPARC og HP Precision arkitekturerne. Den producerede kode er en størrelsesorden bedre end hvad der bliver produceret af oversættere genereret af de klassiske systemer designet af Mosses, Paulson og Wand. Vores maskinsprog behøver ikke dynamisk type-check og er derfor mere realistisk end de som tidligere er blevet betragtet i beviser for oversætter-korrekthed. Vi bruger udelukkende algebraiske specifikationer; beviser gives i den initiale model. Brugen af aktions-semantik gør de maskinlæsbare sprog-specifikationer lette at forstå og behagelige at arbejde med. Vi anser vores oversætter generator for at være et lovende første skridt mod bruger-venlig og automatisk generering af realistiske og bevisligt korrekte oversættere.

I det følgende uddybes beskrivelsen af de opnaede resultater.

Det er et langsigtet mål, for de der arbejder med oversætter generering og korrekthed, at konstruere en *sprog-designer's arbejdsbænk*, med henblik på støtte af sprog-design processen. Hovedkomponterne i sådan en arbejdsbænk er:

- Et *specifikations-sprog* hvis specifikationer er lette at vedligeholde, og tilgængelige uden kendskab til den underliggende teori; og
- En *oversætter generator* som genererer realistiske og bevisligt korrekte oversættere ud fra sådanne specifikationer.

Med sådan en arbejdsbænk kan sprog-designeren:

- Dokumentere design beslutninger;
- Eksperimentere med det nye sprog efter en ændring; og
- Aflevere en oversætter til programmørerne umiddelbart efter designet er færdigt.

Hidtidigt arbejde har ikke formået at bevise korrektheden af nogen realistisk oversætter generator. Derimod er det lykkedes i flere projekter at generere oversættere, der tåler sammenligning med kommercielt tilgængelige oversætere. Disse projekters' erfaring eq at det bliver lettere at generere gode oversætere, hvis:

- En sprog definition ikke indeholder en fuldstændig implementations model; og
- Specifikations-sproget er rettet mod sprog, som traditionelt implementeres via en oversætter.

Vi har designet, implementeret og bevist korrektheden af en oversætter generator, som potentielt kunne blive en komponent i en sprog-designer's arbejdsbænk. Sum specifikations-sprog benytter vi Mosses' *aktions-notation*, som blev designet netop med henblik på at undgå at et sprog-design fastlægger en implementations model. Den centrale komponent i oversætter generatoren er en oversætter fra aktions-notation til absolut kode for en abstrakt RISC maskine. Når oversætter-generatoren gives en sprog-definition i aktions-notation, så bliver den sammensat med aktions-oversætteren, hvilket tilsammen giver en korrekt oversætter for det definerede sprog, idet vi har bevist, at aktions-oversætteren er korrekt.

Aktions-notation er ikke umiddelbart rettet mod at blive implementeret via en oversåtter. Vi har derfor designet en delmængde af notationen, i hvilken man skal skrive alle sprog-definitioner, der gives som input til oversætter generatoren, Denne delmængde er tilstr{aekkelig udtryksfuld til at tillade beskrivelse af sprog med konstruktioner som komplicerede kontrolstrukturer, blok struktur, ikke-rekursive abstraktioner, såsom procedurer og funktioner, og et statisk type system.

Vores semantik af aktions-notationen er en såkaldt naturlig semantik, som vi har specificeret i Mosses' meta-notation for *unified algebras*. Denne semantik er direkte afledt fra den strukturelle, operationelle semantik, hvormed Mosses definerer notationen.

Valget af naturlig semantik som definition af aktions-notationen skyldes at vi så kan benytte en variation af Despeyroux's teknik til bevis af oversætter korrekthed. Denne teknik er blevet anvendt pa en oversætter fra et funktionelt sprog til et idealiseret maskinsprog. I lighed med hvad der benyttes i andre korrektheds-beviser for oversættere, så behøver Despeyroux's maskinsprog dynamisk type-check. En undtagelse herfra er Joyce's teknik, som dog kun er set anvendt på en oversætter af while-programmer.

Det er en alvorlig svaghed ved de tidligere bevis teknikker, at de er baseret på brug af et idealiseret maskinsprog, som behøver dynamisk type-check. Dynamisk type-check letter bevisførelsen kraftigt men betyder længere køretid.

Det er ønskeligt, at også implementationen af det idealiserede maskinsprog bliver bevist korrekt. Det har som konsekvens, at de dynamiske type-check kan ikke umiddelbart udelades af implementationen af det idealiserede maskinsprog, selvom det sprog, der giver anledning til maskinkoden, er statisk type-checket. Det skyldes, at statisk type analyse af de idealiserede maskinsprog ikke synes gennemførlig. Istedet må man så bevise, at i hvert fald de maskinkode programmer, som genereres af den konkrete oversætter, er korrekte. Dette bevis må nødvendigvis involvere både oversætteren til maskinsproget, og implementationen af maskinsproget. Dette er ikke attraktivt, da det ødelægger modulariseringen af det samlede korrekthedsbevis: korrektheden af implementationen af maskinsproget er ikke længere uafhaengigt af andre oversættere. Det er tænkeligt, at den samlede oversættelse, ligeså let kunne bevises korrekt i eet skridt.

De dynamiske type-check kan undgås ved at benytte et maskinsprog, hvis programmer *alle* er type-korrekte. Vi har benyttet et abstrakt RISC maskinsprog, hvor den eneste type er "heltal". Det er således umuligt at se på en given værdi, om man bør opfatte den som en adresse i program teksten, en lager adresse, eller en repræsentation af en sandhedsværdi, et tal, etc. Vi indsætter *ikke* type information i repræsentationen af værdier; der er *ingen* type information til stede pa kørselstidspunktet.

Vores maskine er skabt med SPARC arkitekturen som forbillede. Den indeholder blandt andet globale registre, status bits, register vinduer, og et "random-access" lager. Endnu et realistisk aspekt er, at når maskinen først er begyndt at køre, så stopper den ikke igen. Bemærk her, at vores semantik af maskinsproget ikke tillader fejl som "bus error". Maskinkode i vores maskine bliver oversat til kode til SPARC og HP Precision arkitekturerne.

Oversættelsen fra aktions-notation til maskinsproget sker i to skridt:

1. Type analyse og beregning af kode størrelse; og
2. Kode generering.

Udviklingen af denne oversætter skete ud fra følgende principper:

- Korrekthed er vigtigere end effektivitet; og
- Specifikation og bevis skal være afsluttet inden systemet implementeres.

Et positivt result er, at implementationen blev hurtigt gennemført, og kun en håndfuld mindre fejl (som var blevet overset i beviset!) skulle rettes, før systemet virkede. Et negativt resultat er, at de generede oversættere producerer kode, som kører mindst to størrelsesordener langsommere end tilsvarende maskinkode programmer produceret af håndskrevne oversættere. Det rammer altså et stykke fra det ideelle mål, nemlig realistiske oversættere. Det er dog stadig en forbedring i forhold til klassiske systemer designet af Mosses, Paulson, og Wand, som er endnu en størrelsesorden langsommere.

Korrekthedsbeviset for aktions-oversætteren bruger følgende teknik for at kunne klare sig uden dynamisk type-check:

> Definér relationerne mellem semantiske værdier i kilde-sproget og maskin-sproget med hensyn til *både* en type og en maskintilstand.

Vi definerer således en operation som givet en værdi V, en maskintilstand M, og en type T, giver den *sort* af kildeværdier, som har type T og er repræsenteret af V og M. For eksempel, så kan et heltal repræsentere en værdi af type "liste af sandhedsværdier" ved at pege i en "heap", hvor listen's komponenter er repræsenteret. I dette tilfælde vi1 vores operation give en sort som indeholder netop den list af sandhedsvædier, når den som argumenter får det nævnte heltal, typen "liste af sandhedsværdier", og heap'en.

Muligheden for at give en sort med flere individer behøves, når man abstraherer med hensyn til en "closure" type. Det er fordi, at hvis to aktioner

er ens pånær med hensyn til navngivning af tokens (de er ens med hensyn til "alpha-conversion"), så vil den oversatte kode for dem blive identificeret.

Hvis man ikke involverer maskintilstanden når semantiske værdier relateres, så må man kræve, at maskinens værdier er "selvindeholdte". For ikke-trivielle sprog er det nødvendigt med flere typer værdier i maskinen, og dermed bliver dynamisk type-check nødvendigt.

Alle vore specifikationer er i Mosses' meta-notation. Den tillader kun, at man udtrykker Horn klausuler. Korrekthedsbeviset er struktureret i en række lemmaer, hvoraf de fleste bevises ved induktion i det antal gange "modus ponens" er blevet anvendt.

Sammenfattende kam siges, at vores oversætter generator er et skridt på vej mod automatisk generering af realistiske og bevisligt korrekte oversættere.

Contents

1 Compiler Generation and Correctness 1

1.1 Previous Approaches 4

1.1.1 Classical Compiler Generation 4

1.1.2 Problems with using Denotational Semantics 5

1.1.3 Other Compiler Generators 6

1.1.4 Compiler Correctness Proofs 10

1.1.5 Problems with relying on Run-time Type-checking . . . 14

1.2 A New Approach 16

2 Action Semantics 19

2.1 An Overview of Action Semantics 19

2.1.1 Actions 19

2.1.2 Data and Dependent Data 21

2.2 A Compilable Subset of Action Notation 21

2.2.1 Design Criteria 22

2.2.2 Abstract Syntax 25

2.2.3 Semantic Entities 25

2.2.4 Semantic Funtions 25

3 The Cantor System 27

3.1 An Abstract RISC Machine Language 27

3.2 Compiling Action Notation 30

3.2.1 Compilation Techniques 30

3.2.2 Representation of Action Semantic Entities 33

3.3 Performance Evaluation 36

4 The Correctness Proof 44

4.1 The Correctness Theorem 44

4.2 The Proof Technique 47

4.2.1 Horn Logic 47

4.2.2 Example Proof 49

4.3 An Overview of the Proof 52

4.3.1 Compiler Consistency 53

4.3.2 Correctness of Analysis 54

4.3.3 Completeness 54

4.3.4 Code Well-behavedness 57

4.3.5 Soundness 57

5 Conclusion 58

A A Compilable Subset of Action Notation 61

A.1 Abstract Syntax 61

A.2 Semantic Entities 62

A.2.1 Commitments 62

A.2.2 Storage 63

A.2.3 States 64

A.3 Semantic Functions 65

A.3.1 Actions 65

A.3.2 Unfolding 70

A.3.3 Tuples 72

A.3.4 Dependent Data 73

A.3.5 Unary Operations 74

A.3.6 Binary Operations 74

A.3.7 Data 75

B A Pseudo SPARC Mashine Language 77

B.1 Abstract Syntax 77

B.2 Semantic Entities 78
B.3 Semantic Functions 79
B.3.1 Programs 80
B.3.2 Instructions 80
B.3.3 Moveables 82
B.3.4 Arguments 82
B.3.5 Registers 82

C Actions to SPARC Compiler **84**

C.1 Compile Time Entities 84
C.1.1 Types 84
C.1.2 Data Types 85
C.1.3 Registers 86
C.1.4 Symbol Tables 87
C.1.5 Miscellaneous 87
C.2 Code Macros 88
C.2.1 Lists and Tuples 88
C.2.2 Committing 90
C.2.3 Bookkeeping 91
C.2.4 Call 92
C.3 Analysis 93
C.3.1 Actions 93
C.3.2 Unfolding 96
C.3.3 Tuples 97
C.3.4 Dependent Data 98
C.3.5 Lookup in Symbol Tables 101
C.3.6 Unary Operations 102
C.3.7 Binary Operations 102
C.3.8 Data 103
C.4 Code Generation 104
C.4.1 Actions 104
C.4.2 Unfolding 113

C.4.3 Tuples 117
C.4.4 Dependent Data 117
C.4.5 Lookup in Symbol Tables 120
C.4.6 Unary Operations 121
C.4.7 Binary Operations 121

D Abstraction of Semantic Entities **123**
D.1 Auxiliary Notation 123
D.2 Abstraction Functions 126

E Lemmas **131**
E.1 Auxiliary Notation 131
E.1.1 Code Placement 131
E.1.2 Semantics of Types 133
E.1.3 Program Execution 135
E.2 Compiler Consistency 139
E.3 Correctness of Analysis 149
E.4 Completeness 156
E.5 Code Well-behavedness 178
E.6 Soundness 186

F Main Theorem **195**

G HypoPL Action Semantics **198**
G.1 Abstract Syntax 198
G.2 Semantic Entities 199
G.2.1 Items 199
G.2.2 Coercion 199
G.3 Semantic Functions 199
G.3.1 Programs 199
G.3.2 Declarations 200
G.3.3 Blocks 201
G.3.4 Statements 201

G.3.5 Expressions 202
G.3.6 Operations 202
G.3.7 Integers 203
G.3.8 Identifiers 203

H The HypoPL Bubble-sort Program 204

I Mini-Ada Action Semantics 220
I.1 Abstract Syntax 220
I.2 Semantic Entities 222
I.2.1 Items 222
I.2.2 Closures 222
I.3 Semantic Functions 223
I.3.1 Program 223
I.3.2 Declarations 224
I.3.3 Formals 226
I.3.4 Formal 226
I.3.5 Formals-In 226
I.3.6 Formal-In 226
I.3.7 Nominator 227
I.3.8 Primitive 227
I.3.9 Statements 227
I.3.10 Block 229
I.3.11 Alternatives 229
I.3.12 Names 230
I.3.13 Name 230
I.3.14 Expressions 230
I.3.15 Expression 231
I.3.16 Binary-Operator 231
I.3.17 Control-Operator 232
I.3.18 Integer 232
I.3.19 Identifier 233

J Mini-Ada Benchmark Programs **234**

K Informal Summary of Action Notation **241**

K.1 Basic Action Notation 241

K.1.1 Actions 241

K.1.2 Dependent Data 244

K.1.3 Data 244

K.2 Functional Action Notation 245

K.2.1 Actions 245

K.2.2 Dependent Data 246

K.2.3 Data 246

K.3 Declarative Action Notation 247

K.3.1 Actions 247

K.3.2 Dependent Data 248

K.3.3 Data 248

K.4 Imperative Action Notation 248

K.4.1 Actions 249

K.4.2 Dependent Data 249

K.4.3 Data 249

K.5 Reflective Action Notation 250

K.5.1 Actions 250

K.5.2 Dependent Data 250

K.5.3 Data 251

K.6 Hybrid Action Notation 251

K.6.1 Actions 251

L Data Notation **253**

L.1 Instant 253

L.1.1 Distinction 253

L.1.2 Partial Order 253

L.2 Truth-Values 253

L.2.1 Basics 253

L.2.2 Specifics 253

L.3 Numbers 254
L.3.1 Naturals 254
L.3.2 Integers 255
L.4 Characters 255
L.4.1 Generics 255
L.5 Strings 256
L.5.1 Basics 256
L.6 Trees 256
L.6.1 Generics 256
L.6.2 Basics 256
L.6.3 Specifics 256
L.6.4 Syntax 257
L.7 Lists 257
L.7.1 Generics 257
L.7.2 Basics 257
L.7.3 Specifics 257
L.8 Sets 258
L.8.1 Generics 258
L.8.2 Basics 258
L.8.3 Specifics 258
L.9 Maps 259
L.9.1 Generics 259
L.9.2 Basics 259
L.9.3 Specifics 260
L.10 Tuples 260
L.10.1 Generics 260
L.10.2 Basics 260
L.10.3 Specifics 260

M Informal Summary of Meta Notation 261
M.1 Vocabulary 261
M.1.1 Symbols 262

M.1.2 Variables 262
M.1.3 Titles 262
M.1.4 Marks 262
M.2 Sentences 263
M.2.1 Terms 263
M.2.2 Formulae 264
M.2.3 Clauses 264
M.2.4 Functionalities 264
M.3 Specifications 265
M.3.1 Basic Specifications 266
M.3.2 Modules 267

N Summary of Variable Restrictions **268**

Index **269**

References **276**

Chapter 1

Compiler Generation and Correctness

A *compiler* translates programs in a source language to programs in a target language. It is said to be *correct* if it translates any source program to a target program with the same behavior as the source program. This thesis presents the design, implementation, and proof of correctness of a compiler *generator*.

Most software is written in so-called *high-level* programming languages. The term high-level refers to the conceptual distance between what can be expressed in the language, and what can be expressed at the machine level of a computer. The implementation of such a language is usually provided by a compiler that translates programs into an executable machine language. Even though high-level languages may be preferred for their expressiveness alone, we may also want the compilation process and the generated machine code to be efficient. This requires sophisticated compilers, and such ones are difficult and time-consuming to get correct. The compiler generator described in this thesis puts bits of such sophistication into all generated compilers, and the associated proof guarantees that the compilers are correct.

Writing a correct compiler requires a definition of the syntax and semantics of the source and target languages. A proof of correctness will be based on the given language definitions, and the proof technique will be depend on the style of definition that has been used. *Generating* a correct compiler requires a definition of a *notation* for defining languages. A proof of correctness will be based on the given notation definitions, but not the language definitions. The latter play no part, because any well-formed definition defines some language—it is always "correct", though it need not be what was intended.

The problem of compiler generation is usually simplified by choosing a particular definition of a specific target language [69]. This reduces the task to merely writing and proving the correctness of a compiler for a notation for defining source languages. Such a compiler can then be composed with a language definition to yield a correct compiler for the language, see figure 1. Compiler generators that operate in this way are often called *semantics-directed* compiler generators. The compiler generator described in this thesis is semantics-directed. It accepts language definitions written using *action notation*, and it outputs compilers that emit code in an abstract RISC machine language. Both action notation and the abstract RISC machine language are defined using the unified metanotation of Mosses.

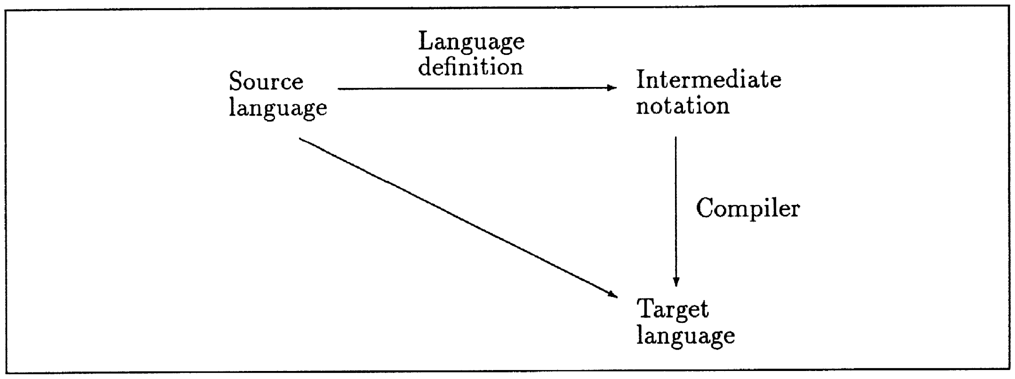

Figure 1.1: Semantics-directed compiler generation.

High-level languages sometimes undergo changes, and new languages regularly get developed. Language definitions are helpful for recording design decisions and design changes, and they may also be helpful if a compiler generator quickly can transform them into implementations. The former requires a flexible and readable notation for defining languages, such that it is easy to figure out where the changes must be made. Action notation is aimed at being flexible and readable, thus hopefully lending a considerable usefulness to the compiler generator described in this thesis.

Automatic generation of correct compilers eliminates the compiler as a source of errors in software. This may be in vain, however, if the implementation of the target language is erroneous. The more high-level a target language is, the more levels of compilation may be involved, and the more errors and inefficiency can be introduced. Recent work in hardware verification indicates that almost realistic machine level architectures can be verified with respect to a low level of the computer, for example the transistor level. By generating compilers that emit code for a verified machine level architecture, a large class of software errors can be eliminated, possibly while retaining efficiency. The abstract RISC machine language which is used as target language in this thesis is not a verified architecture. It is patterned after the SPARC architecture, however, which is born with a semi-formal definition, and which seems to be realistic to verify, because of its simplicity.

One of the long-term goals of work with compiler generation and correctness is the construction of a *language designer's workbench*, envisioned by Pleban [35], for drastically improving the language design process. The major components in such a workbench are:

- A *specification language* whose specifications are easily maintainable, and accessible without knowledge of the underlying theory; and
- A *compiler generator* that generates realistic and correct compilers from such specifications.

With such a workbench, the language designer can:

- Document design decisions;
- Experiment with the new language after a change has been made; and
- Ship a compiler to programmers immediately after the design is finished.

In the following section we examine the major previous approaches to compiler generation and compiler correctness proofs. Various deficiencies will be high-lighted, and criteria for improvements will be expressed. Section 1.2 then gives an overview of the approach taken in this thesis, and it states the major contributions.

Throughout the thesis, the reader is assumed to be familiar with algebraic specification [19], compilation of block structured languages [93], the notion of a RISC architecture [81], and natural semantics [31]. The reader is also assumed to familiar with at least the basic principles of action semantics, see for example the introduction by Mosses [53].

1.1 Previous Approaches

We will examine each of the previous approaches to compiler generation by focusing on:

- The *accessibility* and *maintainability* of the involved specifications;
- The *quality* of the generated compilers; and
- Whether *correctness* has been proved.

These criteria decide whether a system could be useful in a language designer's workbench.

The previous approaches to compiler correctness proofs will be examined to see if an existing proof technique could be helpful in this thesis.

1.1.1 Classical Compiler Generation

The traditional approach to compiler generation is based on *denotational semantics* [76]. Denotational descriptions are written in lambda notation, which in its pure form has a grammar with just three productions. In practice, however, plenty of auxiliary notation is employed, yielding a notation of considerable complexity [51]. Any lambda expression may be seen as specifying an element of a Scott-domain. Alternatively, it can be read as a program which can be executed by repeated use of beta-reduction [3]. Given an implementation of lambda notation, we can then construct a compiler generator which composes any denotational semantics with the implementation of lambda notation. Examples of existing compiler generators based on this idea include Mosses' Semantics Implementation System (SIS) [44], Paulson's Semantics Processor (PSP) [68, 69], and Wand's Semantic Prototyping System (SPS) [89]. In SIS, the lambda expressions are executed by a direct implementation of beta-reduction; in PSP and SPS they are compiled into SECD and Scheme code, respectively. There are no considerations of the possible correctness of either the implementation of beta-reduction, the translations to SECD or Scheme code, or the implementation of SECD or Scheme. The target programs produced by these systems have been reported to run at least three orders of magnitude slower than corresponding target programs produced by handwritten compilers [35].

After these systems were built, several translations of lambda notation into other abstract machines have been proved correct. Notable instances are the categorical abstract machine [11] and the abstract machines that can be derived systematically from an operational semantics of lambda notation, using Hannan's method [26, 24, 25]. It remains to be demonstrated, however, if a compiler which incorporates one of them will be more efficient than the classical systems. Also, the correctness of implementations of these abstract machines has not been considered. Hope for improved efficiency may also come from the implementation techniques based on graph reduction [28].

A major disadvantage of all the abstract machines used as targets for translating lambda notation is that they perform *run-time* type-checking. This may not be significant in comparison with other computational overheads in these machines, but, in general, *compile-time* type-checking is preferable because it allows the generation of more efficient code. It also adds to the complexity of correctness proofs, however, because now the type-checker must be proved correct, as well.

1.1.2 Problems with using Denotational Semantics

It appears that the poor performance characteristics of the classical compiler generators do not simply stem from inefficient implementations of lambda notation. Mosses observed that denotational semantics intertwine model details with the underlying conceptual analysis [46]. Pleban and Lee further observed that not only a human reader but also an automatic compiler generator will have difficulty in recovering the underlying analysis from a denotational semantics [71].

Attempts to recover useful information from lambda expressions include Schmidt's work on detecting so-called single-threaded store arguments and stack single-threaded environment arguments [75, 77]. A successful outcome of such analysis would allow the generated compilers to emit code that operates as usual on a store and a stack, and to use a conventional symbol-table at compile-time. It remains to be seen in practice, though, how much this approach can improve the classical compiler generators, and how generally applicable it is.

An other attempt to analyze lambda expressions is the binding time analysis of Nielson and Nielson [59], currently implemented in the PSI-system [56]. Such analysis allows the generated compiler to make clever decisions about which computations it can carry out at compile-time, and which computations it should generate code for and thus defer to run-time. A code-generator based in these ideas has been proved correct [57]; the quality of the generated code, however, seems to be no better than that generated by compilers generated by the classical systems.

Despite the attempts to compile-time analyze lambda expressions, it seems unlikely that the performance characteristics of compiler generators based on denotational semantics soon will be improved beyond that of existing such systems. Furthermore, denotational semantics is recognized to be neither flexible nor readable, see for example the discussions by Mosses [46], and Pleban and Lee [71]. Le us therefore examine some other approaches to compiler generation.

1.1.3 Other Compiler Generators

The CAT system developed by Schmidt and Völler [78, 79] is aimed at generating compilers for Pascal, C, Basic, Fortran, and Cobol. The notation, called CAT, for defining these languages is a simplification of the union of all their syntactic constructs. This makes CAT itself into a high-level language which has its applicability as language definition notation limited to only little more than the five languages under consideration. The translation of CAT programs proceeds by first doing some optimizing program transformations, then translating into a machine-independent machine language, and finally compiling that into an executable machine language, called CAL. Backends for a range of machine languages have been implemented, including that of MC68000. The generated compilers are of good quality; both compilation and the generated machine code is roughly as good as that of commercially available compilers. The CAT language does not have a formal definition, thus making correctness considerations impossible. Its translation into CAL, however, is formally specified in VDM's metalanguage, META-IV [6]. The executable compiler is manually derived from this specification.

The compiler generator of Kelsey and Hudak [32] is another example of a system that generates compilers of a quality that compares well with commercially available compilers. The system has been used to generate compilers for Pascal, Basic, and Scheme; the compilers generate code for the MC68020 processor. The notation used for defining languages is a call-by-value lambda notation with data and procedure constants and an implicit store. This makes the approach less general that the approaches based on the pure lambda notation, in that it is biased towards a specific style of architecture. Compilation proceeds in six steps that all perform transformation on the intermediate program. After the final step, the program is in one-to-one correspondence with an executable machine language program. Although the syntax of the intermediate language doesn't change during the transformations, the semantics changes radically along the way. There has not been given any specification of the semantics, however, and the compilation process has not been formally specified either. This makes correctness considerations impossible.

A radically different approach to compiler generation is taken by Dam and Jensen [13]. They consider the use of natural semantics [31] (which they call "relational semantics") as the basis of a compiler generator. Giving a natural semantics of a language amounts to specifying a collection of first-order Horn clauses. Thus, a compiler for Prolog, or any other implementation of Horn logic, could be the essential ingredient of a compiler generator. Instead, they devise an algorithm for transforming a natural semantic definition into a compiling specification. The algorithm requires a language definition to satisfy some conditions; it is sufficiently general to apply to a language of while-programs, but has not been implemented. The generated compilers emit code for a stack machine; the correctness of these compilers has been sketched, whereas the implementation of the stack machine is not considered.

It has been known for more than a decade that partial evaluation has an intimate connection to compilation. Partial evaluation produces a residual program from a source program and some of its input data. When given the remaining input data, the residual program yields the same result as the source program would if it was given all the input data. A partial evaluator will accomplish compilation, when it is given as input an interpreter for a language and a program to be compiled, written in that language. The target code will be in the language in which the interpreter is written. If the partial evaluator is self-applicable, then we can apply it to itself and an interpreter for a language. This yields the automatic generation of a compiler. We can even go further and apply the partial evaluator to itself *and* itself. This yields the automatic generation of a compiler generator. The Ceres system of Jones and Tofte [85] is an early example of this, demonstrating that even compiler generators can be automatically generated. Ceres uses a language of flowcharts with an implicit state as the notation for defining source languages.

Another notable partial evaluator is the Similix of Bondorf and Danvy [7, 8] which treats a subset of Scheme. Gomard and Jones implemented a self-applicable partial evaluator, called mix, for an untyped lambda notation [22]. An essential ingredient is an algorithm for binding time analysis. It has been used to generate a compiler for a language of while-programs. The generated compiler emits programs in lambda notation. The correctness of the compiler generator has been proved; it remains to be seen, however, if this approach will lead to the generation of compilers for conventional machine architectures.

The Mess system developed by Pleban and Lee [70, 36, 72, 35] was created as a reaction to the lack of separation between conceptual analysis and model details that is found in the classical compiler generators. Instead of denotational semantics, the approach to defining languages is *high-level* semantics. High-level semantics is compositional, but it does not have a standardized core notation, as does denotational semantics; it is rather a particular style of specification that is advocated. This style involves a notion of *actions*, akin to and inspired by the actions found in precursors of action semantics. A high-level semantic definition involve essentially only compile-time objects; the run-time objects are then used in the definition of the notation for actions. This separation is the key to the success of the Mess system. It has been used to generate a compiler for a non-trivial imperative language. The compiler emits code for the iAPX80286 processor and compares well with for example the Turbo Pascal compiler. High-level semantics has been given a denotational semantics, and the translation of actions is automatically generated from a formal specification. A proof of correctness of this translation has not been given, however. In any case, it is discouraging that such a proof must be given afresh for each new language because new actions often have to be introduced and defined. Lee expresses the hope that proofs can be carried out separately for each language construct; the implementation and the proof of its correctness can then be reused for other languages. The possibility of reusing language definition modules is a central ingredient in the notion of a Language Designer's Workbench, envisioned by Pleban.

The SAM system developed by Pierre Weis [91] is based on essentially the same approach as that taken by Pleban and Lee. The notation used for defining languages is a language of semantic operators. This notation is compiled into code for an abstract machine which in turn is translated into

Designer of the system	Specification language	Quality of generated compilers	Correctness Proof
Mosses	Denotational Semantics	Poor	No
Paulson	Denotational Semantics	Poor	No
Wand	Denotational Semantics	Poor	No
Schmidt and Völler	Amalgamation of five languages	Good	No
Kelsey and Hudak	Lambda notation with implicit store, etc.	Good	No
Pleban and Lee	High-level semantics	Good	No
Gomard and Jones	Denotational Semantics	Poor	Yes

Figure 1.2: Existing Compiler Generators.

machine code. A proof of correctness of this translation has not been given, however. The system has been used to generate compilers for CAML and a Pascal-like language. The code emitted by the generated CAML compiler compares well with that emitted by a handwritten CAML compiler. The code emitted by the generated "Pascal"-compiler runs four times slower than that emitted by a handwritten Pascal compiler.

A summary of the examination of the existing compiler generators is given in figure 1.2. Two things can be concluded, as follows. Firstly, correctness proofs have not been given for any realistic compiler generator. Secondly, better performance of the generated compilers seems to be obtained when:

- Some model details are omitted from a language definition; and
- The notation for defining languages is biased towards "compilable languages".

The lack of correctness proofs limits the confidence we can have in a generated compiler. The approach to correctness taken in this thesis is to focus the attention on semantics-directed compiler generation. Then, we can direct effort to proving the correctness of a compiler from some language definition notation to a specific target language. We have chosen action notation as the language definition notation. Action notation was designed to avoid model details in language definitions; it is defined by a "Plotkin-style" operational semantics. A later chapter presents a "compilable subset" of action notation. Let us now examine the major previous approaches to compiler correctness proofs, to see if an existing proof technique suits our purpose.

1.1.4 Compiler Correctness Proofs

The seminal paper by McCarthy and Painter [37] on correctness of a compiler for arithmetic expressions established a paradigm for proving compiler correctness. This paradigm involves, as summarized by Joyce [30]: abstract syntax; idealized hardware; abstract specification of the compiler; denotational source language semantics; operational target language semantics; correctness stated as a relationship between the denotation of a program, and the execution of its compiled form; and finally, proof by induction on the structure of source language expressions. The correctness statement can be pictured as a commuting diagram, see figure 1.3. Compiler correctness proofs within this paradigm includes those of Milne [40], Stoy [82], and Nielson and Nielson [57].

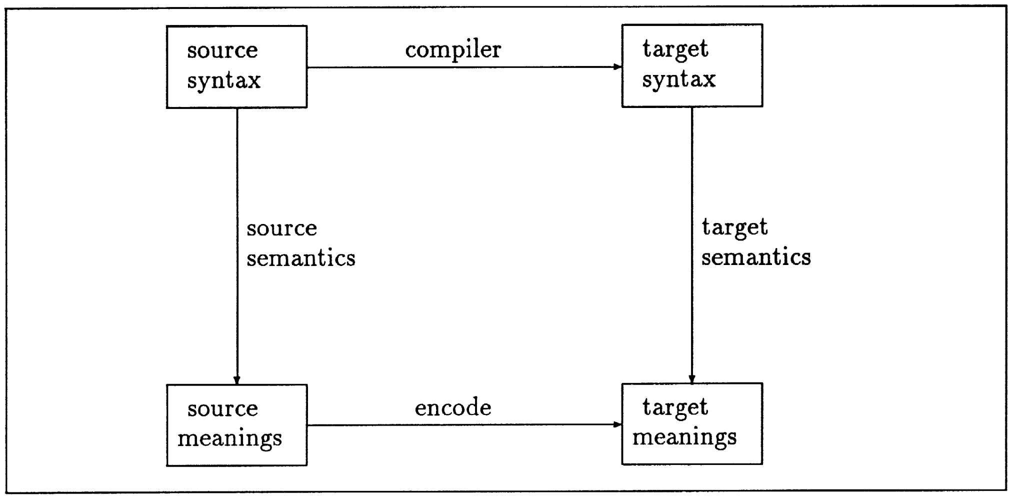

Figure 1.3: Compiler correctness.

Using algebraic methods, the structural induction can be moved into the meta-theory; the correctness proof is then modularized into cases based on the abstract syntax of the source language, see the papers by Burstall and Landin [9], Morris [43], Thatcher, Wagner, and Wright [84], Berghammer, Ehler, and Zierer [4], and also the paper by Mosses [45] where a compiler for an algebraically specified precursor of action notation is proved correct.

It was soon realized that structural induction is not sufficiently powerful if the source language contains for example while loops where the same code may be executed several times. Various improvements have been suggested; they all amount to introducing a times. notion of proof by induction in the length of a computation.

Polak [74] demonstrated that the use of abstract syntax and an abstract compiling algorithm is an unnecessary simplification. He verified a complete compiler implemented in Pascal for a Pascal-like language. This included the verification of a scanner, parser, and static checker, in addition to a code-generator. Part of the verification was automatically performed by the Stanford Verifier.

It has later been demonstrated that complete proofs of compiler correctness can be automatically checked. Two significant instances are Young's [95] work, using the Boyer-Moore theorem prover, and Joyce's [30, 29] work using the HOL system. In both cases, the target code of the translation is a non-idealized machine-level architecture whose implementation has been verified with respect to a low level of the computer, see for example [27, 42]. The verification of both architectures has even been automatically checked. These examples of systems verification [5] are important: they minimize the amount of distrust one need have to such a verified system. Of course, one can still suspect errors in the implementation of the gate-level of the computer, or in the implementation of the theorem prover, but many other sources of errors have been eliminated.

The language considered by Young contains boolean-, integer-, character-, and one dimensional array-types. It has "if" and "loop" control-structures, and procedures. The semantics of the language is given in Boyer-Moore logic. It contains a major cheat: it employs a clock argument to ensure that all computations terminate. This is an artifact that is only present for the purpose of proving correctness, and indeed Young explains that calculating an appropriate lower bound of this clock argument is one of the most difficult aspects of the correctness theorem. The key to why the proof technique works is that programs *cannot* receive input at run-time. If that is possible, as it is in the action notation considered in this thesis, then the proof strategy employed by Young is useless.

The language considered by Joyce is much simpler; it is a language of while-programs. The semantics is essentially a denotational semantics (without cheats!), presented in higher-order logic. It is not clear, however, if his proof technique would generalize to a language with for example block structure, abstractions, or static typing.

The use of denotational semantics renders difficult the specification of languages with non-determinism and parallelism. Such features can be specified easily, however, by adopting the framework of structural operational semantics [73]. For a survey of recent work on proving the correctness of compilers for such languages, see the paper by Gammelgaard and Nielson [17], which also contains a detailed account of the approach taken in the ProCoS project, where the language considered is occam2. In this thesis, we will simply avoid non-determinism and parallelism, and focus on other constructs.

Action notation is defined by a structural operational semantics [54]. In a later chapter we present a subset of action notation without non-determi/-nism and parallelism, and we can then give this subset a special form of structural operational semantics, called natural semantics [31]. In natural semantics, one considers only steps from configuration to *final* states. When both the source and target languages have a natural semantics, then there is hope for proving the correctness of a compiler using the proof technique of Despeyroux [14]. As with the proof techniques used when dealing with denotational semantics, Despeyroux' technique amounts to giving a proof by induction on the length of a computation.

Despeyroux considers an applicative subset of ML, called Mini-ML. The target language is the categorical abstract machine, abbreviated CAM [11]. To give an introduction to the style of specification and proof used by Despeyroux, we will go into a few details of her approach.

The natural semantics of Mini-ML is expressed as $\rho \vdash_{sem} e : \alpha$. It should be read as "the expression e evaluates to the value α in the environment ρ". The natural semantics of CAM is expressed as $s \vdash_{cam} c : s'$. It should be read as "the code c transforms the stack s into s'". The translation from Mini-ML to CAM is expressed as $\bar{\rho} \vdash_{comp} e \to c$. It should be read as "the expression e is translated to the code c, using the symbol-table $\bar{\rho}$". The symbol-table $\bar{\rho}$ is obtained from the environment ρ by removing all values, leaving only names. The values used in the semantics of Mini-ML and those used on the

Completeness:

$$\begin{tikzcd} e \arrow[r, "\text{comp}"] \arrow[d, "\text{sem}"'] & c \arrow[d, dashed, "\text{cam}"] \\ \alpha \arrow[r, "\text{t}"'] & \beta \end{tikzcd}$$

Soundness:

$$\begin{tikzcd} e \arrow[r, "\text{comp}"] \arrow[d, dashed, "\text{sem}"'] & c \arrow[d, "\text{cam}"] \\ \alpha \arrow[r, dashed, "\text{t}"'] & \beta \end{tikzcd}$$

Figure 1.4: Despeyroux's version of compiler correctness.

stack of CAM are different; the transition system $\vdash_t t(\alpha) = \beta$ defines how a Mini-ML value α is represented by a CAM value β.

This setup matches that used in the denotational approaches, see figure 3. Correctness of the compiler is stated differently, though, see figure 1.1.4. In this figure, solid arrows indicate the given facts, and dotted arrows indicate what is to be proved. Both diagrams in figure 4 assume that an expression e has been translated to the code c. The first diagram can be read as follows. If e evaluates to a value α which is represented by a value β, then c produces β on top of the stack. This property is called "completeness":

- **Completeness:** if the source program terminates, then so does the target program, and with the same result.

The second diagram can be read as follows. If c produces a value β on top of the stack, then there exists a value α which is represented by β, such that e evaluates to α. This property is called "soundness" :

- **Soundness:** if the target program terminates, then so does the source program, and with the same result.

Despeyroux proves the correctness statement by induction in the length of those inferences that are assumed to hold in the two diagrams. A central lemma states that the code c for an expression behaves in a disciplined way:

if c terminates, then it produces a value on top of the stack. We call this property "code well-behavedness". We defer to the following chapter a discussion of the treatment of recursion.

We will use a variation of Despeyroux's technique, adapted to the framework of unified algebras.

1.1.5 Problems with relying on Run-time Type-checking

A major deficiency of all the previous approaches to compiler correctness, except that of Joyce, is their using a target language that performs *run-time* type-checking. The following semantic rule is typical for these target languages:

$$(\mathit{FIRST}, \langle v_1, v_2 \rangle : S) \rightarrow v_1 : S$$

The rule describes the semantics of an instruction that extracts the first component of the top-element of the stack, *provided* that the top-element is a pair. If not, then it is implicit that the executor of the target language halts the execution. Hence, the executor has to do run-time type-checking.

Relying on run-time type-checking vastly simplifies the proof task. The reason is that the assertion "if the target program terminates" (which for example Despeyroux used in the statements of "soundness" and "code well-behavedness") then implies "there has been no type errors".

For example, if the instruction *FIRST* is known to terminate, then it is certain that its execution started in a configuration where the top-element of the stack indeed was a pair. This is in marked contrast to a non-idealized machine, where code may by accident execute fine in spite of "logical type errors" and reach the end of the code as if nothing wrong had happened. In other words, the assertion "if the target program terminates" is true too often, making the naive "soundness" and "code well-behavedness" statements false, hence useless.

Run-time type-checking also imposes an unwelcome penalty on execution time because more work has to be done by the executor of the target language. It may be argued, though, that the executor can rely on the source language being statically type-checked, and thus avoid the run-time type-checks. This, however, presents problems for proving correctness of the *executor's* implementation in a non-idealized machine, as explained in the following.

If the executor of the target language does not perform run-time type-checking, then its correctness can only be assured for those target programs that statically are deemed correct. Unfortunately, static type-analysis seems to be difficult for most of the target languages used in previous compiler proofs.

One possibility then is to prove that the executor is correct only for programs obtained by compiling a type-correct program in the source language. This, however, means that we would obtain an unwelcome *coupling* of the source and target languages, preventing in practice the target language from being an independent product, for general use. It also means that the correctness proof for the executor has to involve the compilation of the source language. Then the modularity of the correctness proof is broken: the correctness proof of the executor is not independent of other compilers. In that case it may be as easy to prove in one step the correctness of the combined translation.

A better possibility is to choose a target language where *all* programs are type-correct. Such languages include those with just one type, for example "lambda term", as in the pure lambda notation, or "32 bit word", as in many commercially available machines. They need no type information in their semantics and no run-time type-checking.

In our opinion, it is important that a proof of compiler correctness has the potential of being used as a lemma in the verification of a language implementation with respect to a low level of the computer. To us, this implies that compilation of a statically typed language should be to a target language where it is manageable to specify which programs are type correct, unless run-time type-checking is acceptable.

This thesis addresses the use of independent, realistic target languages without type information in the semantics. Our concern can be sloganized as follows:

> If "well-typed programs don't go wrong", then it should be possible to generate correct code for an independent, realistic machine language that does not perform run-time type-checking.

Note that although Joyce manages without run-time type-checking, he considers only a language of while-programs, and it is not clear how to generalize his approach, as mentioned before. This thesis demonstrates how to state and prove correctness of a compiler to a realistic target language.

Finally note that, in general, run-time type-checking decreases the amount of trust we can have to a system. If the program is used in a safety-critical application, such as a nuclear power-station or medical equipment, then we may want to avoid the possibility of run-time type errors altogether. This thesis considers a target language where *no* run-time checks are performed.

1.2 A New Approach

The previous approaches to compiler generation lack correctness proofs. Assistance is available from work on compiler correctness, but only if one accepts a target language with run-time type-checking or a source language of little more than while-programs.

This thesis overcomes these problems. We have designed, implemented, and proved the correctness of a compiler generator, called Cantor, that accepts action semantic descriptions of programming languages. The generated compilers emit absolute code for an abstract RISC [81] machine language without run-time type-checking. The considered subset of action notation is suitable for describing imperative programming languages featuring:

- Complicated control flow;
- Block structure;
- Non-recursive abstractions, such as procedures and functions; and
- Static typing.

For examples of language descriptions that have been processed by Cantor, see appendices G and I. The abstract RISC machine language can easily be assembled into code for existing RISC processors. Currently, there are assemblers to the SPARC [39] and the HP Precision Architecture [61].

The technique needed for managing without run-time type-checking in the target language is the following:

> Define the relationships between semantic values in the source and target languages with respect to *both* a type and a machine state.

Thus, we define an operation which given a target value V, a machine state M, and a type T will yield the *sort* of source values which have type T and are represented by V and M. For example, an integer can represent a value of type truth-value-list by pointing to a heap where the list components are represented. In this case, our operation will yield a sort containing precisely that truth-value-list, when given the integer, the type "truth-value-list", and the heap.

The possibility of yielding a sort containing several individuals is needed when abstracting with respect to a closure type. This is because if two actions differs only in the naming of tokens (they are equal with respect to "alpha-conversion"), then the compiled code for them will be identical.

In contrast to our approach, for example Nielson and Nielson [58] do *not* involve the machine state when relating semantic values. Instead, they require target values to be "self-contained". Hence, they need to have several types of target values and a target machine that does run-time type checking.

With our approach we can make do with just *one* type of target values, namely integer, thus avoiding run-time type-checking and getting close to the 32-bit words used in the SPARC. Note that we do *not* insert type tags in the run-time representations of source values; *no* type information is present at run-time.

The relationship between semantic values allows the proof of a lemma expressing "code well-behavedness" which is essential when reasoning about executions of compiled code. The required type information is useful during compilation, too; it is collected by the compiler in a separate pass before the code generation. This pass also collects the information needed for generating absolute, rather than relative, code.

The development of Cantor was guided by the following principles:

- Correctness is more important than efficiency; and
- Specification and proof must be completed before implementation begins.

As a result, on the positive side, the Cantor implementation was quickly produced, and only a handful of minor errors (that had been overlooked in the proof!) had to be corrected before the system worked. On the negative side, the generated compilers emit code that run at least two orders of magnitude slower than corresponding target programs produced by handwritten compilers. This is somewhat far from the goal of generating realistic compilers, but is still an improvement compared to the classical systems of Mosses, Paulson, and Wand where a slow-down of three orders of magnitude has been reported [35].

The specification and proof of correctness of the Cantor system is an experiment in using the framework of unified algebras, developed by Mosses [50, 43, 49]. Unified algebras allows the algebraic specification of both abstract data types and operational semantics in a way such that initial models of the specified Horn clauses are guaranteed to exists. So-called constraints can be used to restrict models to the initials ones (and more generally, which we do not exploit).

This thesis demonstrates that also a non-trivial compiler can be elegantly specified using unified algebras. In comparison with structural operational semantics and natural semantics, we replace inference rules by Horn clauses. The notational difference is minor, and only superficial differences appear in the proof of theorems about unified specifications. Where Despeyroux [14] could prove lemmas by induction in the length of inference, we instead adopt an axiomatization of Horn logic and prove lemmas by induction in the number of occurrences of "modus ponens" in the proof in the initial model.

In the following chapter we give an overview of action semantics and the subset of action notation that we compile. In chapter 3 we present the Cantor system, including performance measurements. In chapter 4 we state the correctness theorem of the Cantor system, and we survey the proof. Finally, in chapter 5 we conclude.

Chapter 2

Action Semantics

Action semantics is a framework for formal semantics of programming languages, developed by Mosses [46, 47, 48, 53, 54] and Watt [55, 90]. It is intended to allow useful semantic descriptions of realistic programming languages. This thesis assesses its usefulness for provably correct compiler generation. The following section gives a brief overview of action semantics, taken from Mosses' book [54]. A subsequent section then presents the subset of action notation that can be used in the Cantor system.

2.1 An Overview of Action Semantics

Action semantics is *compositional*, like denotational semantics. It differs from denotational semantics, however, in using semantic entities called *actions*, rather than higher-order functions. The action notation is designed to allow comprehensible and accessible descriptions. Action semantic descriptions scale up smoothly from small example languages to realistic languages, and they can make widespread reuse of action semantic descriptions of related languages.

2.1.1 Actions

Actions reflect the gradual, stepwise nature of computation. A performance of an action, which may be part of an enclosing action, either

- *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or

- *escapes*, corresponding to exceptional termination (the enclosing action is skipped until the escape is trapped); or
- *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- *diverges*, corresponding to nontermination (the enclosing action also diverges).

The information processed by action performance may be classified according to how far it tends to be propagated, as follows:

- *transient*: tuples of data, corresponding to intermediate results;
- *scoped*: bindings of tokens to data, corresponding to symbol tables;
- *stable*: data stored in cells, corresponding to the values assigned to variables;
- *permanent*: data communicated between distributed actions.

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden temporarily. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance, and are unaffected by subsequent divergence, failure, or escape.

The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time:

- the *basic* facet, processing independently of information;

- the *functional* facet, processing transient information (actions are *given* and *give* data);
- the *declarative* facet, processing scoped information (actions *receive* and *produce* bindings);
- the *imperative* facet, processing stable information (actions *reserve* and *unreserve* cells of storage, and *change* the data stored in cells); and
- the *communicative* facet, processing permanent information (actions *send* and *receive* messages, and offer *contracts* to *agents*).

The various facets of an action are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect any bindings. An action may also process finite representations of self-referential bindings, and it can be non-deterministic.

2.1.2 Data and Dependent Data

The information processed by actions consist of items of *data*, organized in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, numbers, characters, strings, lists, sets, and maps. It can also include entities such as tokens and cells, used for accessing other items. Actions themselves are not data, but they can be incorporated in so-called abstractions, which are data, and subsequently 'enacted' back into actions.

Dependent data are entities that can be *evaluated* to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage and buffer. Evaluation cannot affect the current information. Data are a special case of dependent data, and they always yield themselves when evaluated.

2.2 A Compilable Subset of Action Notation

For the purposes of this thesis, we have designed a subset of action notation which is amenable to compilation. The syntax and semantics are presented in appendix A. This section explains the details and discusses some design decisions.

2.2.1 Design Criteria

The development of our subset of action notation was guided by the following criteria:

- It must be given a natural semantics, to make the chosen proof technique applicable;
- It must be monomorphically and statically typed, to avoid type inference and to make code generation simple; and
- It must have a simple semantics, to make definitions and proofs manageable.

These decisions have some immediate consequences:

- Constructs for interleaving and parallelism have to be left out: they cannot be easily described with natural semantics.
- Polymorphic constants must be avoided. Instead, we explicitly constrain the hype of otherwise polymorphic constants; for example we write 'empty-list & [integer] list' instead of just 'empty-list'. The notion of type is added on top of the otherwise untyped language of actions. This development parallels moving from an untyped to a typed lambda notation.
- Actions must be what corresponds to "stack single-threaded" in Schmidt's terminology. We choose an "almost" context-free subset of those, for simplicity of description.
- The construct for unfolding actions must be tail-recursive, to allow simple type analysis and code generation.

We furthermore have to avoid self-referential bindings, needed for example in the description of recursive procedures. The reason for this is rather subtle; it hinges on the expressiveness of the unified meta-notation, as explained in the following.

A self-referential binding is a cyclic structure; the run-time representation will obviously also be cyclic. In Despeyroux' paper, such cyclic structures are represented as graphs with self-loops—both in the source and target languages. This allows her to uniquely determine the run-time representation of a self-referential environment.

Compared to Despeyroux, this thesis uses a much more low-level target language where values can be placed in more than one place in the memory. This means that not only can one target value represent more than one source value, as in Despeyroux' paper, but also is it possible for one source value to be represented by different parts of the memory. In other words, there is no *functional* connection between source and target values; there is only a *relation* stating which source values are represented by a given part of the memory.

In the case of cyclic structures, the relation between semantic values seems to be impossible to define in the unified meta-notation. This is because the meta-notation only allows the expression of Horn clauses. Evidence for this is found in Amadio and Cardelli's paper on subtyping recursive types [2]. They axiomatize several relationships between cyclic structures, and it seems that a rule of the following non-Horn kind cannot be avoided:

$$(x \mathsf{R} y \Rightarrow \alpha \mathsf{R} \beta) \Rightarrow \mu x.\alpha \mathsf{R} \mu y.\beta$$

Since we want to apply the unified meta-notation exclusively in all specifications, we avoid self-referential bindings. Later, the *theorems* about our specifications will be stated in a more expressive notation, namely first-order logic.

A summary of how our subset of action notation relates to full action notation is given in figure 2.1. The subset is sufficiently expressive to describe imperative programming languages featuring complicated control flow, block structure, non-recursive abstractions, such as procedures and functions, and static typing. For examples of language descriptions, see appendices G and I.

The subset is not sufficiently expressive to allow the easy description of for example functional and object-oriented languages, as discussed in chapter 5.

Omissions: Interleaving, parallelism, communication, polymorphic constants, and self-referential bindings.

Basic restrictions:

- Unfoldings must be tail-recursive.
- In 'A or A'', the alternative A is performed first.

Restrictions of the functional facet:

- The types available are truth-values, integers, cells, abstractions, and lists.
- If an action can complete in more than one way then the respective types of the produced data must correspond.
- If an action can escape in more than one way then the respective types of the produced data must correspond.
- The type of the data received by an unfolding must correspond to the type of the data received by the enclosed unfold.
- If 'A and then unfold' occurs in the body of an 'unfolding', then A must not produce data.

Restrictions of the declarative facet:

- If an action can complete in more than one way then the token, type, and order of produced bindings must correspond.
- Actions must be stack single threaded.
- Abstractions can only be closures, must not produce bindings, and cannot be sent out of their defining scopes.

Restriction of the imperative facet:

- Only truth-values and integers are storable.

Figure 2.1: A compilable subset of action notation.

2.2.2 Abstract Syntax

The abstract syntax in appendix A.1 covers roughly half of the full action notation, some of the constructs not in their full generality, though. Two constructs, 'batch-send' and 'batch-receive', are even not standard actions. They allow a primitive form of communication with batch-files, as in standard Pascal [92], and could of course be encoded in action notation, if desired. It would be straight-forward to allow more data-types, such as sets and maps, but we have not bothered to do so. For examples of language descriptions using this subset of action notation, see appendices G and I. For an informal summary of action notation and the accompanying data notation, see appendices K and L. For an informal summary of the meta-notation used throughout, see appendix M. The appendices K, L, M are taken from a draft of Mosses' book [52]. (Appendix L is only an outline; for full details, see [52]).

2.2.3 Semantic Entities

The semantic entities in appendix A.2 differ somewhat from those used by Mosses in his semantics of full action notation. The notion of a final state of an action performance is modeled by a 'state' which can be either 'completed', 'escaped', or 'failed'. Such a state may contain 'data', 'bindings', 'storage', 'input-output', and 'commitment'. The storage component is a mapping from cells to either truth-values, integers, or the special value 'uninitialized'. The commitment component is a truth-value which will be used to express whether the action has committed to the current alternative. If an action has committed to the current alternative, then a subsequent failure does not lead to trying some other alternative, i.e., back-tracking.

Note that we use the data notation for truth-values, integers, and lists, defined in Mosses' book. All components in our lists must have the same type, though3 as in functional languages like ML [41] and Miranda [86].

2.2.4 Semantic Funtions

The semantic functions in appendix A.3 differ from those given by Mosses by being in a natural semantics style [31], rather than a structural operational semantics style [73]. Our semantics has been systematically derived from that of Mosses [54]. In Mosses' definitions the semantics of actions is specified by a function 'run _'. Any combination of arguments to our semantic function final _ _ _ _ _' can easily be transformed into an argument to 'run _', by, say, an operation '$\mathsf{transform}$ _ _ _ _ _'. Similarly, any result of 'run _' can be projected into a result of 'final _ _ _ _ _', by, say, an operation '$\mathsf{project}$ _'. We can then state consistency of the two semantics as follows:

(1) (1) $\mathsf{transform}\ A\ t\ b\ s\ io = arg : arg$;

(2) $\mathsf{run}\ arg = fin : fin$

$\Rightarrow$ $\exists m : \mathsf{state}$

(3) $\mathsf{final}\ A{:}\mathsf{Act}\ t{:}\mathsf{data}\ b{:}\mathsf{bindings}\ s{:}\mathsf{storage}\ io{:}\mathsf{input\text{-}output} = m$;

(4) $\mathsf{project}\ fin = m$

(2) (1) $\mathsf{transform}\ A\ t\ b\ s\ io = arg : arg$;

(2) $\mathsf{final}\ A{:}\mathsf{Act}\ t{:}\mathsf{data}\ b{:}\mathsf{bindings}\ s{:}\mathsf{storage}\ io{:}\mathsf{input\text{-}output} = m{:}\mathsf{state}$

$\Rightarrow$ $\exists fin : fin$

(3) $\mathsf{run}\ arg = fin$;

(4) $\mathsf{project}\ fin = m$

The proof would be by induction in the length of inference, using the technique explained by Nielson and Nielson [60]. We do not give the proof, though.

The occurrence of '*arg*:*arg*' merely restricts '*arg*' to individuals.

Note that the use of commitments builds in a notion of single-threadedness. This is because the specification satisfies the property (proved later) that if the performance of an action does not commit, then the storage and input-output are unchanged.

Note also that the unfolding of actions is described *without* the use of cyclic structures, for example self-referential bindings. This makes it possible to prove correctness of the compilation of unfolding of actions.

Chapter 3

The Cantor System

Our compiler generator accepts action semantic descriptions. It is called Cantor because its main component "*C*ompiles *A*ction *N*otation *TO* *R*isc code". This chapter presents the machine language, the compiler, and some performance measurements of the implementation.

3.1 An Abstract RISC Machine Language

The machine language is patterned after the SPARC architecture; it is called Pseudo SPARC. The syntax and semantics are presented in appendix B. The syntax is *not* presented as a set of strings, because we do not want to parse machine language programs. Rather, we specify instructions as operations. The syntax differs somewhat from the real SPARC syntax. This is because we want to emphasize that the semantics of some instructions has been simplified compared to the real counterparts. The Pseudo SPARC machine language contains 14 instructions that operate on the following machine state:

sparc-state = (program, program-counter, was-zero, was-negative, globals,
windows, memory)

'program' is a mapping from linenumbers to instructions. Alternatively, we could have used a *tuple* of instructions, but the resulting specifications get less readable in our opinion. 'program-counter' is a linenumber, and 'was-zero' and 'was-negative' are status-bits (truth-values). 'globals' models the global registers, and 'windows' models a non-overlapping version of the SPARC register-windows. Finally, 'memory' models six separate "pages" of the main memory, as a mapping from page-identifications to pages. A page is a mapping from addresses (natural numbers) to integers. For example, one of the pages is used as a stack, another as a heap.

The only data manipulated by this language are integers. This means that it is impossible to see from a given data value if it should be thought of as a pointer to an instruction in the program, as an address in the memory, or as modeling a truth-value, an integer, etc.

The uniformity of the data values makes the Pseudo SPARC language more realistic than those considered in previous compiler proofs. It contains two major idealizations, however, as follows:

- **Unbounded word and memory size:** The data values are *unbounded* integers and this requires unbounded word size. We also assume that the program and memory sizes, the number of of registers in a register window, and the number of register windows are unbounded.

- **Read-only code:** The program is plated separately, not in 'memory'. This implies that code will not be overwritten, and that data will not be "executed".

These idealizations simplify the correctness proof considerably, without removing any of the difficulties that we address.

Figure 3.1 shows the 14 Pseudo SPARC instructions and how they (approximately) can be expanded to real SPARC instructions. In practice, the expansion has to take care of fitting instructions using large integers into several real SPARC instructions. It also has to insert additional "nop" instructions into so-called "delay slots". Pseudo SPARC instructions can also be expanded to instructions for the HP Precision Architecture, though with a little more difficulty.

The function that models one step of computation is defined as follows:

- step _ :: sparc-state $\longrightarrow$ spare-state ($total$) .

- step m= next ((program of m) at (program-counter of m) default skip) m .

'step _' models the loading of the current instruction, followed by its execution. The operation 'next _ _ ' is defined in the following style (we give only a single example):

Pseudo SPARC	Real SPARC
skip	`sub %g0, %g0, %g0`
jump Z	`jmpl` Z, `%g0`
branchequal Z branchlessthan Z	`be` Z `bneg` Z

call
return
	`jmpl global, %r8`
	`jmpl %r8 + 8, %g0`

store $R1$ in $R2$ Z P load $R1$ Z P into $R2$	`st` $R1$, $R2 + Z + P$ `ld` $R1 + Z + P$, $R2$
storeregisters loadregisters	`save` `restore`
move RI to R move sum $R1$ RI to $R2$ move difference $R1$ RI to $R2$	`or %g0, RI, R` `add` $R1$, RI, $R2$ `sub` $R1$, RI, $R2$
compare R with RI	`subcc` R, RI, `%g0`

Figure 3.1: The Pseudo SPARC machine language.

- next _ _ :: instruction , spare-state $\longrightarrow$ spare-state ($total$) .
- next call $(p, pc, cz, cn, g, w, q) =$
  (p, g at global default 0, $cz, cn, g,$ update w (map of return-address to pc),
  q) .

Here, 'global' is one of the global registers, and 'return-address' is a user-inaccessible register in the register-window. The use of 'default' models that all registers and memury addresses are initialized to 0 before execution starts. Likewise, the program area contains 'skip' instructions everywhere before the program is loaded.

Note that 'step _' and 'next _ _' are total functions. This emphasizes that computation continues infinitely, once started. For example, the 'call' instruction will be executed even though the global register contained a value that we thought of as a truth-value! It also means that we have avoided alignment problems, etc., so that a typical run-time error such as "bus error" will not occur. This is accomplished by having a word- rather than byte-oriented definition of the Pseudo SPARC machine.

3.2 Compiling Action Notation

The compiler from action notation to Pseudo SPARC machine code is specified in appendix C. This section explains the compilation techniques and the representation of action semantic entities.

3.2.1 Compilation Techniques

The compiler proceeds in two passes:

1. Type analysis and calculation of code size; and
2. Code generation.

For each pass there is a function defined for every syntactic category. Those defined for 'Act' have the following signatures (we simplify a little bit here, to improve the readability):

```
(1)  a-count _ _ _ :: ACT, data-type, symboltable →
          (natural, truth-value, data-type, truth-value, data-type, block)

(2)  perform _ _ _ _ _ _ _ _ _ _ _ _ ::
          Act, data-types, general-register, frozen, symbol-table,
          cleanup, cleanup, cleanup, linenumber,
          linenumber-complete, linenumber-escape, linenumber-fail →
          (program, general-register, general-register) .
```

Action notation contains constructs, e.g., 'escape', 'fail', that are at a slightly lower conceptual level than those found in the high-level programming languages that action notation is suitable to describe. Thus, the definition of the type analysis and code generation employ unusual techniques, though not very difficult. For example, the definition of 'perform' requires as argument both the desired start-address ('linenumber') of the code to be generated, but also addresses of where to jump to, should the performance complete ('linenumber-complete'), escape ('linenumber-escape'), or fail ('linenumber-fail'). These addresses are calculated using 'a-count' which, in addition to type analysis, calculates the size of the code to be generated.

The function 'a-count' is defined as a forwards abstract interpretation, computing with types of tuples of data ('data-type'), types of bindings ('symbol-table'), and code sizes ('natural'). The first 'truth-value' component tells if the action being analyzed has a chance of completing. If it does, then the following 'data-type' component tells the type of the tuple of data that will produced. The next two components give similar information about escaping.

The type checking of an action is performed using the operations 'compare-data-types _ _ _ _' and 'compare-blocks _ _ _ _' which ensure that the types of data and bindings produced by two branches of an action such as '_ or _' are equal.

We mentioned in an earlier chapter that our context-free syntax for actions is not completely "stack single-threaded". We need a compile-time check that ensures that closures are not sent out of their defining scopes as components of data. This check is performed by the operation 'abstraction-free _' and is inserted in the analysis of the constructs '$\llbracket$ $\llbracket$"furthermore" A $\rrbracket$ "hence" A' $\rrbracket$' no and '$\llbracket$ $\llbracket$"furthermore" A $\rrbracket$ "thence" A' $\rrbracket$'.

Note that '$\llbracket$' and '$\rrbracket$' are *syntactic* node-constructing brackets; nesting indicates tree structure.

The analysis of unfolding of actions requires the computation of a fixed point. This is because the data-type and symbol-table produced by the analysis of an unfolding depend on themselves. Fortunately, we can compute this fixed point in one step, because the unfoldings in our subset of action notation are tail-recursive.

The function 'perform' takes as arguments the 'data-type' and 'symbol-table' that are also supplied to 'a-count'. In addition, it takes a 'general-register' which at run-time will contain a pointer to a representation of the tuples of data that will be received when executing the code. The set 'frozen' contains those registers that the code to be produced must not modify, and the three 'cleanup' values are natural numbers that indicate how much to pop from the stack, should the performance complete, escape, or fail.

The calculation of whether an action can complete or not, and whether it can escape or not, are examples of the compile time analyses that are built into the compiler. They are used to generate better code, and they are fully integrated in the proof of correctness, see later.

It is *not* the case that "if the type analysis succeeds, then so does the code generation". The reason is the checks of the form 'either(e is empty-list, u_n is 0) = true', see for example C.4.1.(17.8). If the information 'u_n is 0' was made available to the type analysis, then we could move the checks to the type analysis, and the the above property would be truce We have not bothered to do so, however, because it has no impact on stating and proving correctness.

As an example of how the compiler works, see the following excerpt from the compiling specification.

```
(1) (1) d-count $D$ $h$ $d$ =($n$: natural, truth-value-type)
    ⇒ a-count ⟦ “check” $D$:Department ⟧$h$ $d$ =
            ac-state sum($n$, 2, e-size, 12) true () false () empty-list .
(2) (1) d-count $D$ $h$ $d$ = ($n$: natural, truth-value-type) ;
    (2) $l'$ = sum($l, n$)
    (3) $l''$ = sum($l'$, 2, e-size) ;
    (4) evaluate $D$ $h$ $a$ $f$ $d$ $l$ sum($l''$, 6) = ($p$:program, $r$:general-register)
    ⇒ perform ⟦ “check” $D$: Department ⟧ $h$ $a$ $f$ $d$ $u_n$ $u_e$ $u_f$ $l$ $l_n$ $l_e$ $l_f$ =
      a-state overlay(
            $p$,
            map of sum($l'$, 0) to ( compare $r$ with 0 ) ,
            map of sum($l'$, 1) to ( branchequal sum($l''$, 6) ) ,
            empty-list-code $r$ sum($l'$, 2)
            putcommit $l''$ 0 ,
            finalize sum($l''$, 3) $u_n$ 0 $l_n$
            putcommit sum($l''$, 6) 0 ,
            finalize sum($l''$, 9) $u_f$ 2 $l_f$ )

            $r$ $a$ .
```

The first definition calculates the size of the code generated by the second definition. It also does the type-checking. The meaning of the action ‘check D’ is to check whether D evaluates to true or false, and it should then “complete” or “fail”, accordingly. The generated code first computes the result of D, and then it does a branchequal, as expected. (We represent true as 1 and false as 0.) This is not all, however. Because of the generality of action notation a lot of additional code is also generated. For example, a commonly found action such as ‘check (it is true)’ yields 37 lines of code. It should be noted, though, that it is this clear structure of the code that made the correctness proof manageable. Section 3.3 includes a discussion of the possibilities for optimizing the code. The operation ‘overlay _’ concatenates the pieces of code given as arguments.

The operations 'empty-list-code _ _', 'finalize _ _ _ _', and 'putcommit _ _' are code macros. Their purpose is connected to the chosen representation of action semantic entities which we consider next.

3.2.2 Representation of Action Semantic Entities

The semantic entities manipulated by actions are 'data', 'bindings', 'storage', 'input-output', and 'commitment'. Their representation in the Pseudo SPARC machine is specified in appendix D. The eleven abstraction functions are defined using the technique explained earlier:

> Define the relationships between semantic values in the source and target languages with respect to *both* a type and a machine state.

For example, the abstraction function for data 't-abs n q p h' takes an address n of a data-representation, a machine state q, a machine language program p and a data-type h as arguments. It yields the sort of data which have type h and are represented at the address n in q and p.

The other ten abstraction functions specify the representation of bindings ('b-abs _ _ _ _' and 'e-abs _ _ _ _'), storage ('store-abs _ _', 's-abs _ _', and 'storable-abs _'), input-output ('i-abs _ _' and 'o-abs _ _'), commitment ('c-abs _'), states ('m-abs _ _ _ _ _ _ _ _ _ _ _'), and individuals of sort datum ('v-abs _ _ _ _').

Informally, the represent at ions are as follows:

- Data tuples are represented as a pointer to a linked list;
- Bindings are represented as a stack in the memory page 'stack', with the stack pointer in the global register 'sp' and the static link pointer in the register 'static-link'. The first element in a stack frame is the static link. The type of an entire stack is a 'symbol-table', that of an individual stack frame is a 'block';
- Storage is represented in the memory page 'storage', with the address of the first free dell represented in the global register 'firstfree';

- The input and output files are represented in the memory pages '`input`' and '`output`', in both cases with the length of the file stored at address 0; and
- Keeping track of the current alternatives requires a stack of individual commitments. This stack is represented in the memory page '`commits`', with the stack pointer in the global register '`cp`'. The code for an action is required to place on top of the commitment stack a value representing the commitment component of the final state of the performance of the action.

The stack of commitments is managed by the three code macros '`putcommit`', '`combine-commit`', and '`combine`', see C.2.2. The first places a commitment on top of the stack. The second performes a logical disjunction on the two top-elements of the stack, removes them, and places the result on top of the stack. The third is used when generating code for the action combinators, as follows.

Consider for example the action combinator 'A then A'', and suppose that the performance of A completes and that the performance of A' either completes, escapes, or fails. The commitment value of the final state of the performance of A must now be combined with that of the final state of the performance of A'. To do this we want to use the code macro '`combine-commit`', and afterwards be able to jump to the right code address, depending on whether A' completed, escaped, or failed.

To be able to do the combination of commitments in a simple way, the code macro '`combine`' contains *three* copies of '`combine-commit`'. If the performance of A' for example completes, then it jumps to the first copy of '`combine-commit`' and then jumps to the address corresponding to completion. Should the performance of A' escape, then it jumps to the second copy of '`combine-commit`' and then jumps to the address corresponding to escape. The pattern is similar if A' fails.

To summarize: at most one of the three copies of '`combine-commit`' in '`combine`' will be executed because they are all followed by a jump instruction.

Individuals of sort datum are represented as follows:

- True as 1 and false as 0;
- Natural numbers as themselves;

- Cells as the cell number;
- Closures as a pair of a code address and a static link pointer; and
- Lists as a pointer to a linked list (the same representation as that of data tuples).

The pairs and linked lists are represented in the memory page 'heap', with the heap pointer in the global register 'hp'. Note that we do not do garbage collection in the heap and storage; that would significantly complicate the correctness theorem and proof.

The manipulation of representations of lists and data tuples is managed by the six code macros 'empty-list-code', 'single-list-code', 'concatenation-code', 'head-code', 'tail-code', and 'at-code', see C.2.1.

The enaction of a closure can lead to completion, escape, and failure, and it is in general undecidable which one will occur. To make things simple we require all code for actions to store a value in the general register 'cef' which tells if the action has completed, escaped, or failed. One of the purposes of the code macro 'finalize' is to take care of this. It is then straightforward to generate code at closure enaction points. Note that the 'cef' register dynamically contains which subsort of 'state' (either 'completed', 'escaped', or 'failed') the machine currently represents.

The code for closure enaction ('call-sequence') is simple because we use a RISC architecture. We need not represent a dynamic chain on the stack because we can instead shift the register window to obtain a fresh register for the static link pointer, a fresh register for storing the return address, and also fresh general registers. When returning from a closure we simply shift the register window back.

Allocation of general registers are handled using the operation 'free-register _'. Given a set f of "frozen" general registers, 'free-register f' yields the register with the lowest possible number that is not in f. A register gets frozen for example if it represents data that can be accessed by both branches of an action, typically '_ and then _'.

The code generation for unfolding of actions, see C.4.1.(16), starts with allocating a fresh register a' and copying to it the representation of the received data, contained in the register a. The register a is then frozen so that the unfolding does not destroy the data it represents—it might be needed later. We can now use the register a' to represent data to be passed to an unfold. This data need not be the same each time the unfold is encountered, hence the need for the extra register a'.

3.3 Performance Evaluation

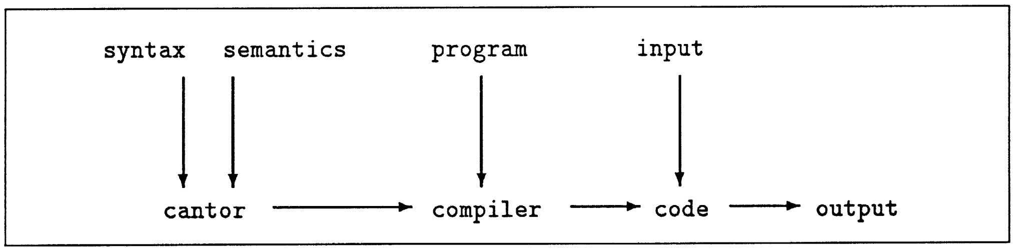

Figure 3.2: The Cantor system.

The Cantor system has the structure shown in figure 3.3. In practice, a session with Cantor looks as follows on the screen:

```
cantor syntax semantics compiler
compiler program code
code input output
```

The compiler generator `cantor` is written in Perl [88], and the generated compilers are written in Scheme [1]. Examples of a syntax and a semantics are given in appendix G and I; it is the LaTeX sources of the appendices that are processed by `cantor`. Any generated compiler contains a syntax checker, a program-to-action transformer, the action compiler described above, and finally a Pseudo SPARC assembler that currently can emit code for the SPARC and the HP Precision Architecture. The input file is a sequence of integers, as is the output file.

The HypoPL language, defined in appendix G, is taken from Lee's book on realistic compiler generation [35], with the difference that we treat nesting of procedures in its full generality but do not allow recursion.

- Generating a compiler for HypoPL takes 3 seconds.

We have used this compiler to translate Lee's bubblesort program (50 lines), see appendix H.

- Compile time: 486 seconds;
- Object code size: 114688 bytes; and
- Object code execution time (for sorting 10 integers): 0.1 seconds.

These figures indicate that the system is rather tedious to work with in practice. Appendix H also presents excerpts from the action and assembly code generated from the bubblesort program.

The Mini-Ada action semantics in appendix I has been the primary benchmark in our experiments with the Cantor system. Mini-Ada is a subset of Ada [15] featuring static typing, constants, variables, one-dimensional array-types, functions and procedures with in and in out (reference) parameters, various control structures, and the usual expressions. Note that the select construct in Mini-Ada can be used as a "case"-statement, and that also the input-output statements (read and write) are non-standard Ada. Otherwise, the Mini-Ada specification is a subset of one given by Mosses in his book [54].

- Generating the Mini-Ada compiler takes 9 seconds.

We have used this compiler to translate a number of benchmark programs, see appendix J and the overview described in figure 3.3. The sieve, euclid, and fib programs contain a main loop that allows iterating the computation. This will be practical when we later compare the object code emitted by the Mini-Ada compiler with that emitted by handwritten compilers.

bubble: Bubblesorts a number of integers (50 lines).

sieve: Performs the sieve of Erathosthenes prime number generator (30 lines).

euclid: Computes the greatest common divisor of two numbers using Euclid's algorithm (20 lines).

fib: Computes the 56'th Fibonacci number (30 lines).

Figure 3.3: The Mini-Ada benchmark programs.

The number of Pseudo SPARC instructions emitted for each benchmark program is given in figure 3.4. When the Pseudo SPARC code is compiled to code for the SPARC, then the size is approximately doubled. A slightly worse blow-up is obtained when compiling to the HP Precision Architecture.

Number of Pseudo SPARC instructions generated:

bubble:	16697
sieve:	12096
euclid:	7386
fib:	9095

Figure 3.4: Object code size.

Unfortunately, we have no access to an Ada compiler that generates code for either of the two architectures that we consider Instead, we have made comparison with the standard C [33] compiler for those architectures. It is perhaps unfair to compare Ada and C, but we still believe that using the C compiler gives a good indication of the capabilities of Cantor. We expect that the C compilers generate better code than potential Ada compilers. Hence, when we compute the slow-down compared to C, we will take it as an upper bound of the slow-down compared to Ada. We of course had to rewrite the Mini-Ada programs slightly to get them accepted by the C compilers. Since the constructs in C are less general than those in Ada, we expect a significantly better performance of the C-generated code, than what could be expected from Ada-generated code.

	C	C^{opt}	Mini-Ada
bubble:	1.0	2.2	542
sieve:	1.2	2.1	377
euclid:	1.1	1.6	136
fib:	1.1	1.7	210

Figure 3.5: Compile times.

Figure 3.5 shows the compile time in seconds when using the C compiler, the C compiler with maximal optimization switched on, and the Cantor-generated Mini-Ada compiler. The timings in this figure were recorded on the SPARC; the compilers run almost equally fast on the HP as on the SPARC.

	C	C^{opt}	Mini-Ada	Slow-down
bubble:	4.4 (1000 numbers)	2.1 (1000 numbers)	0.9 (37 numbers)	149
sieve:	1.3 (400 itera.)	0.4 (400 itera.)	1.2 (1 itera.)	369
euclid:	5.4 (30000 itera.)	0.9 (30000 itera.)	0.8 (30 itera.)	148
fib:	1.2 (10000 itera.)	0.2 (10000 itera.)	0.8 (36 itera.)	185

Figure 3.6: Object code execution time on the SPARC.

	C	C^{opt}	Mini-Ada	Slow-down
bubble:	7.2 (1000 numbers)	4.7 (1000 numbers)	4.3 (37 numbers)	436
sieve:	1.2 (400 itera.)	0.4 (400 itera.)	4.5 (1 itera.)	1500
euclid:	4.5 (30000 itera.)	4.4 (30000 itera.)	2.7 (30 itera.)	600
fib:	1.1 (10000 itera.)	0.5 (10000 itera.)	3.9 (36 itera.)	985

Figure 3.7: Object code execution time on the HP Precision Architecture.

Figures 3.6 and 3.7 show the object code execution time in seconds for the benchmark programs. They also show the estimated slow-down when using the Mini-Ada compiler, compared to the C compiler *without* optimization. The slow-down factors were computed by simple extrapolation. The figures indicate, unsurprisingly, that the Mini-Ada-generated code runs faster on the SPARC than on the HP. This is because the Pseudo SPARC machine language was designed to match the SPARC instructions, not the HP instructions. Thus, more code is generated for each Pseudo SPARC instruction when compiling to the HP.

The performance of the object code is most fairly compared on the SPARC. Taking the differences of C and Ada into account, we conclude that the object code run at least two orders of magnitude slower than corresponding code produced by handwritten Ada compilers.

This is somewhat disappointing but still an improvement compared to the classical systems of Mosses, Paulson, and Wand where a slow-down of three orders of magnitude has been reported [35].

Inspection of the code emitted by Cantor-generated compilers reveals that the inefficiency mainly stems from three sources:

- Lack of compile time constant propagation;
- Poor register allocation; and
- Naive representation of bindings, closures, and lists.

(Constant propagation covers possible attempts to follow the flow of commitments). To illustrate this and to explain further details of the action compiler, we will analyze the code generated for a particular action. The action to be considered, see below, is part of the semantics of the HypoPL bubble-sort program, see appendix H.

- execute ⟦ “write” ⟦ “ −” 999 ⟧ ⟧ =

```
        | give negation 999
        then
        | give the given (truth-value | integer) #1 or
        | give the (truth-value | integer) stored in the given cell #1
        then
        |  batch-send it
```

Appendix H also contains the seven(!) pages of code generated for this action: they will be analyzed below.

First note the occurrence of the following subaction.

-
```
| give negation 999
then
| give the given (truth-value | integer) #1 or
| give the (truth-value | integer) stored in the given cell #1
```

The part following 'then' appears because the semantics of the HypoPL statement 'write E' uses the auxiliary action combinator 'coercively _' when evaluating the expression E. This is just one possible style for writing the semantics, however, another is used in the semantics of Mini-Ada, without 'coercively _'. The latter style would not yield the second part of the above action, and would thus be an optimization in itself.

A further example that illustrates the superiority of the style used in the Mini-Ada semantics (efficiency-wise at least) is shown in appendix H. It is the action generated for the printNums procedure of the HypoPL bubble-sort program. In that action, the second part of the above action appears nine times. If we had used the style from the Mini-Ada semantics, then five of the nine copies would be avoided.

Note also that if the compiler could perform constant propagation, then code need only be generated for an action corresponding to 'batch-send negation 999'.

Consider now the Pseudo SPARC code generated for ⟦ "write" ⟦ "-" 999 ⟧ ⟧. First appears the code for 'negation 999'. This involves placing the value 999 in a general register, and then computing the negation of it by placing the value 0 in a global register and finally computing the difference of the contents of the two registers.

Then appears the code for giving that value. This involves inserting it into a list (six instructions), placing on top of the commitment stack the value 0, representing 'uncommitted' (three instructions), removing no values from the stack, since the performance is not leaving a scope of bindings (one instruction that could be dropped), placing in the global register 'cef' the value 0, representing completion (one instruction), and finally jumping to the start address of the code for the action following 'then'.

Actually, six more instructions are generated for 'give D'. They are used in case D evaluates to 'nothing' so that the performance fails. If the compiler could analyze that for some occurrences of 'give D', it is certain that D yields 'nothing', then the code could be optimized, and similarly if it certain that it does not yield 'nothing'. In the specific case we are considering, the last six instructions could be dropped.

After the code for 'give' follow two instructions, generated from 'then', that are used if performance of 'give' escapes. The first instruction moves the contents of the register which represents the data produced on escape to another register. The latter register is the same as the one where data will be placed if the action *following* 'then' escapes. In other words, if the combined action escapes, then data produced are placed in that register. The second instruction jumps to the place in the code where the second occurrence of 'then' handles escapes.

Note that if the compiler could analyze that the first argument to 'then' cannot escape, then the two instructions could be dropped.

Combining the optimizations for 'give' and 'then' would enable a further optimization. This is because the instruction that jumps to the start of address of the code for the action following 'then' in that case would be a jump to the following line of code!

The code for the action following the first occurrence of 'then' starts with the code for 'the given (truth-value | integer) #1'. The type-check has occurred at compile time, so the code merely has to extract the first component of the representation of the received tuple of data, and place it in a register. This is done by placing the value 1 in a generalregister, and then using one of the general code macros (twelve instructions) for accessing a linked list. This could of course be optimized, and further optimization was possible if the represent ations of the components of the received data tuple was placed directly in registers.

After the code for 'give' follows the code generated from 'A or A'' that is used to check whether A has completed, escaped, or failed, and, in the first two cases, move data representations to other registers before jumping to appropriate code addresses. If the performance of A fails, then the following code checks whether the performance has committed or not. If so, it jumps to the next occurrence of 'or' (where something similar will happen). If the performance has not committed, then performance of A' begins. If the compiler could analyze that in this case A cannot commit, then we could optimize the code.

The code for the action following 'or' has benefited from a compile-time analysis. The compiler has type-checked 'the given cell #1' and found out that it must yield 'nothing'. Hence, the whole actions must fail, so only six instructions need to be generated, namely those to which the code macros 'putcommit' and 'finalize' expand. If the compiler could analyze that the action before 'or' cannot fail, then the code for the action following 'or' could be omitted altogether.

After the code for the second argument to 'or' appear 18 instructions to which the code macro 'combine' expand. Again, if the compiler could analyze the value of commitments, then this code could be omitted.

We will not analyze the seven pages of code further. We have indicated several possibilities for compile-time analyzing actions and for using the computed information to improve the code generator. Improving the action compiler in this way, however, would significantly complicate the correctness theorem, which we consider next.

Chapter 4

The Correctness Proof

This chapter states the correctness theorem of the Cantor system, outlines the proof technique to be used, and gives an overview of the proof.

4.1 The Correctness Theorem

To give an overview of the correctness theorem, see appendix F, we will introduce a bit of notation as follows (we simplify a little bit, to improve the readability):

(1) run _ _ _ :: Act, [integer] list $\rightarrow$ state

(2) sparc-run _ _ _ :: program, natural, page $\rightarrow$ sparc-state .

(3) compile _ :: Act $\rightarrow$
  (program,truth-value, data-type,
  truth-value, data-type,
  general-register, general-register) .

(4) abstract _ _ _ _ _ _ _ :: spare-state, truth-value, data-type, truth-value,
  data-type, general-register, general-register $\rightarrow$ state .

(5) ibs _ _ :: natural, page $\rightarrow$ [integer] list .

(6) (1) a-count A () (list of empty-list) $=$ ac-state n z_n h_n z_e h_e empty-list ;
  (2) perform A () (reg 0) empty-set (list of empty-list) 0 0 0 0 n n n $=$
    a-state p a_n a_e
  $\Rightarrow$ compile A:Act $=$ result p z_n h_n z_e h_e a_n a_e .

We have only given the definition of 'compile _', in terms of 'a-count' and 'perform'. The operations have the following informal meaning:

1. The operation 'run A il' specifies the performance of an action A which is given the empty tuple of data, no bindings, an empty-storage, an empty output-file, and the input-file il (an integer-list). If the performance terminates, then that will result in a final state ('state') which can be either completed, escaped, or failed.

2. The operation 'spare-run p n se' specifies loading the program p into the program area, and then taking n steps starting in line 0. It also records if the execution at any point "jumps outside the code". The memory, registers, status bits, and output file are initialized appropriately, the input file is initialized to se. 'spare-run' is defined in terms of 'step', described in chapter 3.

3. The operation 'compile A' translates the action A into a machine language program p and it also gives type information about what will be produced when performing A. The program p will start in line 0.

4. The operation 'abstract m_p h_n z_e h_e a_n a_e' will give a *sort* including all those states (from the action-level) that are represented by the spare-state m_p, and that have the type expressed by the following four arguments. The last two arguments are those registers which will contain pointers to the representations of the data produced, should the action complete or escape.

5. The operation 'i-abs n se' will give the input-file ('[integer] list') which is represented by the natural number n and the page se.

The use of both type information and a machine state in the definition of 'abstract' makes it possible to make do without type information in the semantics of Pseudo SPARC.

None of the above five operations are total. The performance of an action may diverge; the execution of a machine program may "jump outside the code"; the compilation of an action may find a type error; the machine state may represent no state at all from the action-level; and the page for input-files may contain something without the right format.

The meta-notation for unified algebras makes it particularly easy to specify such partial operations. This is because it supports a *unified* treatment of sorts and individuals: an individual is treated as a special case of a sort. Thus operations can be applied to sorts as well as individuals. A vacuous sort represents the lack of an individual, in particular the 'undefined' result of a partial operation. For example, if the performance of the action A with input-file il terminates, then 'run A il' will be an individual, otherwise it will be a vacuous sort. We need not specify explicitly that such sorts are vacuous; if it does not follow from the specification that they contain an individual, then they will automatically be vacuous.

The operations 'run', 'sparc-run', 'compile', and 'i-abs' will all yield either an individual or a vacuous sort. In contrast, 'abstract' may yield a sort containing *several* individuals, and it may also yield a vacuous sort. The possibility of yielding a sort containing several individuals is needed when abstracting with respect to a closure type. This is because if two actions differs only in the naming of tokens (they are equal with respect to "alpha-conversion"), then the compiled code for them will be identical.

We can now state the correctness theorem. Note that 't :- s' is merely another syntax for 't : s'.

Theorem:

(1) compile A: Act = (p:program z_n:truth-value h_n:data-type z_e:truth-value h_e:data-type a_n:general-register a_e:general-register) ;

(2) i-abs (se at 0) se:page = il:[integer] list

$\Rightarrow$ (1) run A il = m_a:state $\Rightarrow$
($\exists$ m_p:sparc-state $\exists$ n:natural .
sparc-run P n se = m_p
abstract m_p z_n h_n h_e a_n a_e :- m_a) ;

(2) sparc-run p n se = m_p:sparc-state $\Rightarrow$
($\exists$ m_a:state .
run A il = m_a
abstract m_p z_n h_n h_e a_n a_e :- m_a) ;

The structure of the theorem resembles the correctness statement of Despeyroux. Informally:

If the action A is compiled into a machine language program p (and some additional type information, etc., is produced), and the input-file il is represented properly in the machine as se, then two properties hold:

1. **Completeness:** If the performance of the action A (with input-file il) terminates in state m_a, then there exists a spare-state m_p and a number n such that an n-step execution of p will reach m_p, and m_p represents m_a (and the program-counter points to the last line of p).
2. **Soundness:** If an n-step execution of p (with input se) reaches m_p (and the program-counter points to the last line of p), then there exists a state m_a, represented by m_p, such that a performance of A (with input il) will terminate in m_a.

Notice that it is built into the definition of 'sparc-run', and hence the correctness theorem, that the execution of the machine language program never "jumps outside the code".

4.2 The Proof Technique

Our approach to correctness can be summarized as follows:

1. Give a natural semantics to both action notation and the abstract RISC machine language;
2. Make the compiling of action notation simple; and
3. Use a variation of Despeyroux's proof technique [14].

This section explains how to adapt Despeyroux's proof technique to the framework of unified algebras. In outline, we adopt an axiomatization of Horn logic and can then prove lemmas in the initial model by induction in the number of occurrences of "modus ponens". As an example, we prove that the semantics of our subset of action notation is singlethreaded.

4.2.1 Horn Logic

Despeyroux expresses natural semantics in the Gentzen's system style, with axioms and inference rules. In such a system one can make natural deduction, and can then prove lemmas about the system by induction in the length of such deductions. In contrast, the framework of unified algebras provides Horn clauses. We can of course simulate Gentzen style axioms with Horn clause axioms, and we can mimic the use of an inference rule

$$\frac{A}{B}$$

with $A' \Rightarrow B'$, where A' simulates A, and B' simulates B. To be able to do deduction, we adopt an axiomatization of Horn logic, see for example [80], as follows.

All specifications in the meta-notation for unified algebras can be transformed into a core notation which is outlined in the following. Let Ω be a so-called homogeneous first-order signature, that is, a pair $\langle \Sigma, \Pi \rangle$ where Σ is a set of operation symbols and Π is a set of predicate symbols. In the setting of unified algebras, it is required that

$$\Sigma \supseteq \{\mathsf{nothing}, \_ \mid \_, \_\&\_\}$$

and

$$\Pi = \{\_ = \_, \_ \leq \_, \_ : \_\}$$

Further, let Γ be a set of Horn clauses built up from Ω as explained in appendix M.2. Any specification Γ of such Horn clauses will be augmented with some basic Horn clauses, stating for example the reflexivity of '$\_ \leq \_$', see [49]. Finally, let F be a formula built up from Ω. We will then write

$$(\Omega, \Gamma) \vdash F$$

(read F is (Ω, Γ)-deducible) if $(\Omega, \Gamma) \vdash F$ can be obtained by finitely many applications of the following deduction rules:

$$\frac{}{(\Omega, \Gamma) \vdash t = t} \qquad \text{(Reflexivity)}$$

$$\frac{(\Omega, \Gamma) \vdash s = t \quad (\Omega, \Gamma) \vdash t = u}{(\Omega, \Gamma) \vdash s = u} \qquad \text{(Transitivity)}$$

$$\frac{\{(\Omega, \Gamma) \vdash s_i = t_i\}_{i=1}^{n}}{(\Omega, \Gamma) \vdash f(s_1, \dots, s_n) = f(t_1, \dots, t_n)} \text{ if } f \in \Sigma \qquad \text{(Functional Congruence)}$$

$$\frac{\{(\Omega, \Gamma) \vdash s_i = t_i\}_{i=1}^{2} \quad (\Omega, \Gamma) \vdash p(s_1, s_2)}{(\Omega, \Gamma) \vdash p(t_1, t_2)} \text{ if } p \in \{\_ \leq \_, \_ : \_\} \qquad \text{(Predicative Congruence)}$$

$$\frac{\{(\Omega, \Gamma) \vdash F_i\}_{i=1}^{n}}{(\Omega, \Gamma) \vdash F} \text{ if } (F_1; \dots F_n \Rightarrow F) \in \Gamma \qquad \text{(Modus Ponens)}$$

A deduction rule consists of a *conclusion* (given beneath the line), none, one, or several *premises* (given above the line), and possibly a *condition* (given at the right-hand side of the line). A deduction rule stands for the statement:

> If all premises are deducible, the condition is satisfied, and F is a formula built up from Ω, then the conclusion $(\Omega, \Gamma) \vdash F$ is deducible.

With these deduction rules, we can do proof by induction in the length of deduction. Throughout, however, we will do the inductions in the number of occurrences of only "modus ponens". We will call this "proof by induction in the length of *inference*", to avoid confusion. Note that a single application of modus ponens corresponds closely to a natural deduction step. This makes our proof strategy close to Despeyroux's.

All lemmas proved by induction in the length of deduction are satisfied by the *initial* model of the specification. The key property of an initial model needed here is that it only contains entities that are values of ground terms (it contains "no junk"). This property makes it possible to exhaust all cases in a proof, as exemplified in the following section.

4.2.2 Example Proof

To illustrate the proof technique, we will prove a fundamental property of our semantics of actions. It is the singlethreadedness lemma in appendix E.4. This lemma states that if an action does not commit, then the storage and

input-output are unchanged.
Lemma: (Singlethreadedness of Actions)

> t : data ;
> b : bindings ;
> s : storage ;
> io : input-output

$\Rightarrow$

(1) (1) final A: Act $t\ b\ s\ io = m_a$: state

> $\Rightarrow$ either(both((storage of m_a) is s, (input-output of m_a) is io),
> (commitment of m_a) is committed) = true ;

(2) (1) unf-final U: Unf ⟦ "unfolding" U': Unf ⟧ $tb\ s\ io = m_a$: state

> $\Rightarrow$ either(both((storage of m_a) is s, (input-output of m_a) is io),
> (commitment of m_a) is committed) = true .

Proof: We need to prove the conjunction of (1) and (2) because the semantic functions for Act and Unf ('final' and 'unf-final') are mutually recursive. We will prove the lemma by induction on the length of inference of formulas of the form (1.1) and (2.1). (Notice that "(1.1)" is a short form of "(1) (1)"). The proof is given in the initial model. This model satisfies a formula if and only if the formula can be deduced. Such a deduction can involve only the finitely many Horn clauses in appendix A, so in the following we can exhaust all cases.

In the base case, we consider inferences that involve only one application of a clause for either final or unf-final. Such inferences exist only for the formula (1.1). They use the clauses A.3.1.(1)—(14),(39)—(40), respectively. We will only give the details of two of the cases, the others are similar.

Firstly, consider the clause for "complete", in A.3.1.(1), that is

- final "complete" $t\ b\ s\ io =$ completed () empty-map $s\ io$ uncommitted .

Here, $m_a =$ completed () empty-map $s\ io$ uncommitted. It follows from the totality of 'completed _ _ _ _ _' in A.2.3.(7) and the lemma's assumptions about s and io being individuals, that m_a is an individual. The conclusion of (1) now follows since both (storage of m_a) is s and (input-output of m_a) is io are true.

Secondly, consider the first clause for $[\![$ "give" D:Dependent$]\!]$, in A.3.1.(6), that is

- (1) evaluated $D\ t\ b\ s = v$: datum

  $\Rightarrow$ final $[\![$ "give" D:Dependent $t\ b\ s\ io =$
  completed v empty-map $s\ io$ uncommitted .

Here, $m_a =$ completed v empty-map $s\ io$ uncommitted. This holds because evaluated $D\ t\ b\ s = v$: datum. Since v is an individual we also in this case get that m_a is an individual. The conclusion of (1) follows as in the first case.

In the induction step, we consider the remaining clauses for 'final' and 'unf-final'. We will give the details of only four of the cases, the others are similar.

Firstly, consider the first clause for $[\![$ "enact" "application" D:Dependent "to" D':Tuple $]\!]$, in A.3.1.(15), that is

- (1) evaluated $D\ t\ b\ s =$ closure-abstraction A:Act D'':Data b':bindings ;

  (2) multi-evaluated $D'\ t\ b\ s = t'$: data
  $\Rightarrow$ final $[\![$ "enact" "application" D:Dependent "to" D':Tuple $]\!]$ $t\ b\ s$
  $io =$ final $A\ t'\ b'\ s\ io$.

Here, also final $A\ t'\ b'\ s\ io = m_a$, using transitivity. Since A, t', and b' are individuals, we can apply the induction hypothesis (1). This immediately yields the conclusion.

Secondly, consider the clause for $[\![$ "unfolding" U:Unf $]\!]$, in A.3.1.(17), that is

- final $[\![$ "unfolding" U:Unf $]\!]$ $t\ b\ s\ io =$ unf-final U $[\![$ "unfolding" U $]\!]$ $t\ b\ s$ io .

Here, also unf-final U $[\![$ "unfolding" U $]\!]$ $t\ b\ s\ io = m_a$, using transitivity. Since U is an individual we can apply the induction hypothesis (2). This immediately yields the conclusion.

Thirdly, consider the first clause for $[\![$ A:Act "then" A':Act $]\!]$, in A.3.1.(21), that is

- (1) final $A\ t\ b\ s\ io =$ completed t' empty-map $s'\ io'\ c'$;

(2) $\mathsf{final}\ A'\ t'\ b\ s'\ io' = \mathsf{completed}\ t''\ b''\ s''\ io''\ c''$

$\Rightarrow$ final $[\![$ A:Act “then” A':$\mathsf{Act}]\!]$tbsio:= completedt”$b''\ s''\ io''$ $\mathsf{either}(c', c'')$

Since t', s', and io' are assumed to be individuals we can apply the induction hypothesis (1) to the assumptions. If either c' or c'' are true, then so is $\mathsf{either}(c', c'')$ and then the conclusion is immediate. If both $'$ and c'' are false, then from the induction hypothesis we get that s is s', io is io', s' is s'', and io' is io'' all are true. The conclusion follows.

Fourthly, consider the clause for “unfold”, in A.3.2.(13), that is

- unf-final “unfold” $[\![$ “unfolding” U:Unf $]\!]$ $t\ b\ s\ io$ = final U $[\![$ “unfolding” U $]\!]$ $t\ b\ s\ io$.

Here, also final $[\![$ “unfolding” U $]\!]$ $t\ b\ s\ io = m_a$, using transitivity. Since U is an individual we can apply the induction hypothesis (1). This immediately yields the conclusion. □

In the proof of the lemma we considered explicitly only six cases, namely those of:

A.3.1.(1) :	“complete” ;
A.3.1.(6) :	$[\![$ “give” D:Dependent$]\!]$;
A.3.1.(15):	$[\![$ “enact” “application” D:Dependent “to” D':Tuple $]\!]$;
A.3.1.(17):	$[\![$ “unfolding” U:Unf $]\!]$;
A.3.1.(21):	$[\![$ A:Act “then” A':Act $]\!]$; and
A.3.2.(13):	“unfold” .

The remaining cases could be treated similarly. In appendix E we prove other lemmas, most of them using the same proof technique as in the proof just given. In these proofs we also consider only a few of the cases, the others are similar.

4.3 An Overview of the Proof

The proof of the correctness theorem is structured into a sequence of lemmas. These lemmas are stated and proved correct in appendix E. The appendix also contains some auxiliary notation that is used in the lemmas. This section explains the lemmas and it emphasizes the difficulties that we meet in the proofs because we deal with a realistic target machine. The validity of the main theorem, see appendix F, is an immediate consequence of the lemmas.

4.3.1 Compiler Consistency

The six lemmas in appendix E.2 state properties of the compiler *without* reference to the semantics of either action notation or the Pseudo SPARC machine code. Instead, they concern the algorithm for allocating free registers, the consistency between the analysis and code generation parts of the compiler, and the type analysis of unfoldings of actions. Here is an overview of the lemmas.

- **Calculation of Free Registers:** This lemma states that the algorithm for allocating free registers behaves as expected: when given a set of "frozen" registers, it yields one which is *not* in that set.
- **Code Macro Size:** This lemma states that the twelve code macros are "well-placed". This means that each of them starts in the desired line, that the instructions are placed consecutively, and that they have the expected size.
- **Compiler Consistency:** This lemma states that the calculation of the size of the code for actions is correct and that the code is "well-placed". It also states part of the "stack single-threadedness" requirement: bindings cannot be produced if the performance immediately afterwards leaves the current scope of bindings.
- **Consistent use of Symbol-tables:** This lemma states that the calculation of the size of the code for accessing the representation of bindings is correct and that the code is well-placed. The size of the code depends on the contents of the symbol-table.

The last three of these lemmas use the auxiliary notation in appendix E.1.1. For example, the notion of "well-placedness" is captured by the predicate 'well-placed _ _ _'. If we had represented machine language programs as tuples of instructions, then the well-placedness predicate could have been slightly simplified, since the instructions then "automatically" are placed consecutively.

The two remaining lemmas concern the type analysis of unfoldings of actions. They use the predicate 'ac-less', see appendix E.1.2, which defines an ordering on the types of final states of actions. The type analysis works by assuming an 'ac-less'-minimal type of the final state of a performance of the unfolding, and then computing a greater type. It thus maps types to types. The lemmas state key properties of this computation.

- **Increasing Type Analysis of Unfoldings:** This lemmas states that if the type analysis of an unfolding succeeds, then it yields a greater type of the final state than the one that it assumed to begin with. This property is exploited in the proof of the following lemma.
- **Type Analysis of Unfoldings computes a Fixed Point:** This lemma states that the type analysis is "idempotent": when it has been successfully performed it has computed a fixed point.

The first four lemmas mentioned in this section are used repeatedly in the proof of several of the remaining lemmas in appendix E. The last two lemmas justify why we do not perform an iterative type analysis of unfoldings.

4.3.2 Correctness of Analysis

The lemma in appendix E.3 states that the type analysis asserts correct typings, relative to the semantics of actions. In particular, the type analysis of unfoldings computes a correct type. The lemma does not refer to the semantics of the Pseudo SPARC machine code. It uses the auxiliary notation in appendix E.l.2. For example, given a 'type', the operation 'vc-a bs _' yields the sort of individuals contained in 'datum' that has that type.

The lemma is not needed in other proofs, but its proof may serve as a gentle introduction to the more complicated proofs later on.

4.3.3 Completeness

The main lemma in appendix E.4 is a strengthened version of the completeness statement used in the main theorem. Moreover, appendix E.4 contains two other lemmas concerning the correspondence between types and representations of action semantic entities, and concerning the single-threadedness of actions. Here is an overview of the lemmas.

- **Sound Semantics of Types:** This lemma states that if a value is represented by a given integer with respect to a type and a memory, then the semantics of the type contains that value.

- **Single-threadedness of Actions:** This lemma states that if the performance of an action does not commit, then the storage and input-output are unchanged.

- **Completeness:** This lemma states that if the source program compiles to a target program, and if the target program starts in a state which represents the arguments to the source program, and if the source program terminates, then an execution of the target program will reach "the end" of the code, and with a result which represents the result of the source program.

The first lemma uses the auxiliary notation in appendix E.1.2, and the third lemma uses the definition of 'spare-final _ _ _ _' in appendix B and the auxiliary notation in appendix E.1.3. The auxiliary notation used by the completeness lemma concerns the execution of Pseudo SPARC machine language programs, here is an overview.

- The operation 'spare-final n m_p lt nw' specifies the execution of n machine program steps starting in the state m_p. If the execution in any step, except the last, jumps outside the code contained in m_p, then the result is 'nothing'. This property means that only a finite code area needs to be used in correct implementations of actions. The execution is required to be such that the program-counter assumes one of the linenumbers in lt only in the last step; otherwise the result is 'nothing'. This property imposes a correspondence between n and lt: the execution reaches "the end of the code" (lt) in exactly n steps. Furthermore, the result state needs to be at a certain register window level, indicated by nw, otherwise the result is 'nothing'. This property means that when a machine program is terminated, then we can return to the operating system via an address in the 'return-address' register (provided that this register has not been overwritten).

- The binary predicate 'n leq nt' specifies that the natural number n is less than all the natural numbers in the tuple nt. This predicate is used to assert that the program-counter has not reached or passed certain points in the program text.

- The operation 'cleaned-up sp cef n u_n u_e u_f' maps an old stack pointer sp to a new one. This computation takes place when an action has terminated. There we need to assert that the machine program has cleaned up the stack correctly. Depending on whether the action has completed, escaped, or failed (recorded in cef), there should be popped either difference(u_n, n), u_e, or u_f elements, respectively. Here, n is the number of bindings that the action has produced. This number can only be non-zero for completing actions.

- The operation 'up-to n' maps a natural number n to the set of natural numbers that are strictly smaller than n. This is useful in assertions about initial segments of memory pages.

- The predicate 'q-earlier n n' q q'' asserts that the two memories q and q' differs only in the stack and heap pages. Furthermore, these two pages have identical initial segments, up to the natural numbers n and n', respectively. The predicate is symmetrical, but we have used the name 'q-earlier' to indicate that the predicate is used to compare memories that occur at two different points of the same program execution.

- The predicate 'm-earlier m_p m'_p' asserts that the two memories m_p and m'_p are identical, except that the former's memory is 'q-earlier' the latter's. Similarly, the predicate 'mq-earlier m_p q' asserts that the memory of m_p is 'q-earlier' than q.

The remaining ten predicates are used as pre- and post-conditions in the statement of the completeness lemma (and also in the later lemmas on code well-behavedness and soundness). The major reason why they are rather complicated is that because we are dealing with a realistic random-access memory, a single update of the memory can destroy the represent ation of some source value. To handle this, we formulate the assumptions of the lemma such that values are not only assumed to be represented by the current memory, but also by any other memory that the current one is 'q-earlier' than. Correspondingly, the conclusions of the lemma state that the result is represented by not only the current memory, but also by those that the current one is 'q-earlier' than.

4.3.4 Code Well-behavedness

The first lemma in appendix E.5 states the program in a machine state is never modified during its execution.

The second lemma states the well-behavedness of code generated from actions; it is used repeatedly during the proof of soundness. The key complication handled by this lemma is that the execution of a machine program does not stop by itself, as explained in the following.

Consider an action A which is compiled to code p', and suppose that A is a subaction of some larger action which is compiled to code p. Suppose further that at some point of the execution of p, the program counter has assumed the start-address of p'. To reason about the execution of p' we need to express what happens when the execution reaches "the end", of p'. But since the execution does not stop at this point, it is not immediate that if it has passed this point, then at some earlier step it actually reached this point (it might "jump over"). The code well-behavedness lemma states that indeed it did, and moreover, it has used the memory and registers in a disciplined fashion, and the machine state will represent an abstract state (with the type given by the compiler).

The code well-behavedness lemma does not refer to the semantics of actions.

4.3.5 Soundness

The soundness lemma in appendix E.6 is a strengthened version of the soundness statement used in the main theorem. It states that if the source program compiles to a target program, and if the target program starts in a state which represents the arguments to the source program, and if an execution of the target program reaches "the end" of the code, then the source program will terminate, and with a result which is represented by the result of the target program.

Chapter 5

Conclusion

Our compiler generator is specified and proved correct solely in an algebraic framework. To our knowledge, it is the first time that this has been accomplished.

The generated compilers emit realistic, albeit poor, machine code. Still, it is significantly better than that produced by the classical systems of Mosses, Paulson, and Wand. Furthermore, the code is absolute, rather than relative. To handle that in the correctness proof, we prove lemmas expressing "compiler consistency".

The proof of correctness demonstrates how to deal with a realistic machine language without type information in the semantics. Our machine language has a random-access memory, and computation continues infinitely, once started. Given an implementation of our machine language, and a correctness proof for it, one could compose both the implementations and the correctness proofs.

Our proof technique is an improvement over the previous ones. The technique of Joyce handles a non-idealized target language, but it has only been shown to apply to a compiler of while-programs. The other proof techniques either rely on the target language being run-time type-checked, or face increased, and to our knowledge unsolved, difficulties in proving the correctness of a target language executor which avoid the run-time type-checks.

The use of action semantics makes the processable specifications easy to read and pleasant to work with. We believe that the Cantor system is a promising first step towards user-friendly and automatic generation of realistic and *correct* compilers. We also consider it to be a step towards a provably correct implementation of a practically useful language designer's workbench. We have illustrated our approach on a non-trivial subset of Ada, hoping to indicate that such a workbench could have been a helpful tool during the design of Ada.

The Cantor system is based on a subset of action notation. Even without changing this subset, future work may take several directions.

- **Better object code:** More compile time analysis should be employed, to improve the code generator.
- **Completely realistic target language:** A target language without the idealizations discussed in this thesis should be used.
- **Faster compiler:** The action compiler should be rewritten in a less functional style than the current one, to get acceptable compile times.
- **Automatic proof check:** The recent advances in automatic proof checking should be exploited, to obtain a very trustworthy system.
- **Polymorphic type inference:** The polymorphic type inference of Even and Schmidt [16] should be used, to avoid the explicit type information in our subset of action notation.

To summarize, we believe that a provably correct and practically useful language designer's workbench is a realistic possibility.

Future work may also attempt to improve the Cantor system by enlarging the used subset of action notation. This may require a more expressive meta notation and a more powerful proof technique. For example, the specification of self-referential bindings seems to require more expressiveness than what is offered by Horn clauses. Also, the specification of parallelism and communication seems to require a structural operational semantics style rather than the natural semantics style that we have used, thus demanding a different proof technique.

Another possibility is to drop the stack single-threadedness requirement, to be able to describe higher-order functions.

We will conclude with a discussion of how much improvement of Cantor is needed to allow the treatment of an increasingly important class of languages: the *object-oriented* languages.

Major examples of object-oriented languages are Simula [12], Smalltalk, [21], C++ [83], Beta [34], Eiffel [38], and Self [87]. They have four significant commonalities: assignments, objects, inheritance, and late binding. They may thus be understood as being imperative languages with three additional constructs.

An object groups together variables and procedures, and is thus akin to a module in Modula-2 [94]. Object-oriented languages, in contrast to Modula-2, allow objects to be stored in variables and to be passed as arguments and returned as results. The semantic treatment of objects, inheritance, and late binding goes beyond the capabilities of the Cantor system, see below.

An object could in the full action notation be represented as a pair of maps: one for the variables and one for the procedures. This would require more data structures than what is present in the Cantor system, and it would also require that we were allowed to store (values containing) abstractions. Finally, the procedures in an object are typically mutually recursive, thus requiring self-referential bindings.

Inheritance may be understood as a mechanism for deriving modified versions of recursive structures. It is thus not surprising that its denotational semantics requires an involved manipulation of fixed points, see the paper by Cook and the author [10] for details. An action semantics of inheritance would require much of the generality of self-referential bindings that is found in the full action notation, but not in the subset that we have used.

Late binding means that a procedure call is *dynamically* bound to an implementation. This makes static typing a challenging problem that currently is the subject of much research, see for examples the papers by the author and Schwartzbach [64, 66, 65, 67]. A treatment of late binding would require the Cantor system to have a more complicated type system.

Appendix A

A Compilable Subset of Action Notation

A.1 Abstract Syntax

needs: Data Notation/Numbers/Naturals .
introduces: token .
grammar:

(1) Act = “complete” | “escape” | “fail” |
“commit” | “diverge” | “regive” |
⟦“give” Dependent ⟧ | ⟦ “check” Dedendent ⟧ |
⟦“bind” token “to” Dependent⟧ |
⟦“store” Dependent “in” Dependent⟧ |
⟦“allocate” (“truth-value” | “integer”) “cell”⟧ |
⟦“batch-send” Dependent ⟧ | ⟦ “batch-receive” “an” “integer”⟧ |
⟦“enact” “application” Dependent “to” Tuple”⟧ |
⟦“indivisibly” Act ⟧ | ⟦ “unfolding” Unf ⟧ | ⟦ Act Infix Act ⟧ |
⟦⟦“furthermore” Act ⟧ (“hence” | “thence”) Act ⟧ .

(2) Unf = ⟦ Act Infix Unf ⟧ | ⟦ Unf “or” Act ⟧ | “unfold” .

(3) Tuple = “()” | Dependent | ⟦ Tuple “,” Tuple ⟧ | “them” .

(4) Dependent = “true” | “false” | natural |
⟦“empty-list” “ & ” “[” Type “]” “list” ⟧ |

⟦“closure” “abstraction” “of” Act “ & ”
“[” “perhaps” “using” Data “]” “act” ⟧ |
⟦ Unary Dependent ⟧ | ⟦ Binary “(” Dependent “,” Dependent “)” ⟧ |
⟦ Dependent (“is” | ⟦ “is” “less” “than” ⟧) Dependent ⟧ |
⟦“component#” Dependent “items” Dependent ⟧ |
“it” | ⟦ “the” “given” Datum “#” natural ⟧ |
⟦“the” Datum “bound” “to” token ⟧ |
⟦“the” Datum “stored” “in” Dependent ⟧ |
⟦“(” Dependent “)” ⟧ .

(5) Infix = ⟦“and” “then”⟧ | “then” | “before” | “trap” | “or” .

(6) Unary = “not” | “negation” | ⟦“list” “of” ⟧ | “head” | “tail” .

(7) Binary = “both” | “either” | “sum” | “difference” | “concatenation ” .

(8) Datum = “datum” | “cell” | “abstraction” | “list” |
⟦ Datum ” | “ Datum ⟧ | Type .

(9) Data = “()” | Type | ⟦ Data “,” Data ⟧ .

(10) Type = “truth-value” | “integer” |
⟦“truth-value” “cell” ⟧ | ⟦ “integer” “cell” ⟧ |
⟦“ [” Type “]” “list” ⟧ .

A.2 Semantic Entities

includes: Data Notation .
needs: Abstract Syntax .

A.2.1 Commitments

introduces: commitment , uncommitted , committed .

(1) commitment = truth-value .

(2) uncommitted = false

(3) committed = true .

A.2.2 Storage

introduces: storage , storage-map ,
cell , truth-value-cell , integer-cell , truth-value-cell _ , integercell _ ,
storable , uninitialized ,
storable in _ , empty-storage .

(1) storage = (storage-map, natural) .

(2) storage-map = [cell to storable | uninitialized] map .

(3) cell = truth-value-cell | integer-cell (*disjoint*) .

(4) truth-value-cell _ :: natural + truth-value-cell (Ma/) .

(5) integer-cell _ :: natural $\rightarrow$ integer-cell (*total*) .

(6) storable = truth-value | integer .

(7) uninitialized : uninitialized .

(8) uninitialized & storable = nothing .

(9) storable in _ :: cell $\rightarrow$ storable (*strict, linear*) .

(10) storable in *ct*:truth-value-cell = truth-value .

(11) storable in *ci*:integer-cell = integer .

(12) (1) *s* :storage = (*m*:storage-map, *n*:natural) ;
(2) *ce*:cell is in mapped-set *m* = true ;
$\Rightarrow$ (*m* at *ce*) : (storable in *ce*) | uninitialized .

(13) empty-storage = (empty-map, 0) .

(14) (truth-value-cell *n*:natural) is (truth-value-cell *n′*:natural) = *n* is *n′* .

(15) (integer-cell *n*:natural) is (integer-cell *n′*:natural) = *n* is *n′* .

(16) *ct*:truth-value-cell is *ci*:integer-cell = false .

A.2.3 States

needs: Commitments , Storage .

introduces: abstraction , closure-abstraction _ _ _ ,
datum , data , bindings , input-output ,
state , completed , escaped , failed ,
completed _ _ _ _ _ , escaped _ _ _ _ , failed _ _ _ ,
storage _ , input-output _ , commitment _ .

t : data ;
b : bindings ;
s : storage ;
io : input-output ;
c : commitment

$\Rightarrow$

(1) closure-abstraction _ _ _ :: Act, Data, bindings $\rightarrow$ abstraction ($total$) ;

(2) datum = truth-value | integer | cell | abstraction | [datum] list ($disjoint$) ;

(3) data = datum* ;

(4) bindings = [token to datum] map ;

(5) input-output = ([integer] list, [integer] list) ;

(6) state = completed | escaped | failed ($disjoint$) ;

(7) completed _ _ _ _ _ ::
data, bindings, storage, input-output, commitment $\rightarrow$ completed ($total$) ;

(8) escaped _ _ _ _ :: data, storage, input-output, commitment $\rightarrow$ escaped ($total$) ;

(9) failed _ _ _ :: storage, input-output, commitment $\rightarrow$ failed ($total$) ;

(10) storage _ :: state $\rightarrow$ storage ($total$) ;

(11) input-output _ :: state $\rightarrow$ input-output ($total$) ;

(12) commitment _ :: state $\rightarrow$ commitment ($total$) ;

(13) storage (completed t b s io c) = s ;

(14) storage (escaped t s io c) = s ;

(15) storage (failed s io c) = s ;

(16) input-output (completed t b s io c) = io ;

(17) input-output (escaped t s io c) = io ;

(18) input-output (failed s io c) = io ;

(19) commitment (completed t b s io c) = c ;

(20) commitment (escaped t s io c) = c ;

(21) commitment (failed s io c) = c .

A.3 Semantic Functions

needs: Abstract Syntax , Semantic Entities .

A.3.1 Actions

needs: Unfolding , Tuples , Dependent Data .

introduces: final _ _ _ _ _ .

• final _ _ _ _ _ :: Act, data, bindings, storage, input-output → state .

t , t' , t'' : data ;
b , b' , b'' : bindings ;
s , s' , s'' : storage ;
io , io' , io'' : input-output ;
m_a : state ;
c' , c'' : commitment

⇒

(1) final "complete" t b s io = completed () empty-map s io uncommitted ;

(2) final "escape" t b s io = escaped t s io uncommitted ;

(3) final "fail" t b s io = failed s io uncommitted ;

(4) final "commit" t b s io = completed () empty-map s io committed ;

(5) final "regive" t b s io = completed t empty-map s io uncommitted ;

(6) (1) (evaluated $D\ t\ b\ s = v$: datum

⇒ final ⟦ "give" D:Dependent ⟧ $t\ b\ s\ io =$
completed v empty-map $s\ io$ uncommitted ;

(7) (1) evaluated $D\ t\ b\ s =$ true
⇒ final ⟦ "check" D:Dependent ⟧ $t\ b\ s\ io =$
completed () empty-map $s\ io$ uncommitted ;

(8) (1) evaluated $D\ t\ b\ s =$ false
⇒ final ⟦ "check" D:Dependent ⟧ $t\ b\ s\ io =$ failed $s\ io$ uncommitted ;

(9) (1) evaluated $D\ t\ b\ s = v$: datum
⇒ final ⟦ "bind" k:token "to" D:Dependent ⟧ $t\ b\ s\ io =$
completed () (map of k to v) $s\ io$ uncommitted ;

(10) (1) evaluated $D\ t\ b\ s = x$: (storable in ce) ;

(2) evaluated $D'\ t\ b\ s = ce$:cell ;

(3) s:storage = (m:storage-map, n:natural)

⇒ final ⟦ "store" D:Dependent "in" D':Dependent ⟧ $t\ b\ s\ io =$
completed () empty-map (overlay(map of ce to x, m), n) io committed ;

(11) final ⟦ "allocate" "truth-value" "cell" ⟧ $t\ b$ (m:storage-map, n:natural) $io =$
completed (truth-value-cell n) empty-map
(overlay(map of n to uninitialized, m), successor n) io committed ;

(12) final ⟦ "allocate" "integer" "cell" ⟧ $t\ b$ (m:storage-map, n:natural) $io =$
completed (integer-cell n) empty-map
(overlay(map of n to uninitialized, m), successor n) io committed ;

(13) (1) evaluated $D\ t\ b\ s = i$ integer ;

(2) io :input-output = (il : [integer] list, ol : [integer] list)
⇒ final ⟦ "batch-send" D:Dependent ⟧ $t\ b\ s\ io =$
completed () empty-map s (il, concatenation(list of i, ol) committed ;

(14) (1) io = (concatenation(list of i:integer, il : [integer] list), ol : [integer] list)
⇒ final ⟦ "batch-receive" "an" "integer" ⟧ $t\ b\ s\ io =$
completed i empty-map s (il, ol) committed ;

(15) (1) evaluated $D\ t\ b\ s =$ closure-abstraction A:Act D'':Data b':bindings ;

(2) multi-evaluated $D'\ t\ b\ s = t'$: data
⇒ final ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ $t\ b\ s\ io =$

final A t' b' s io ;

(16) final $[\![$ "indivisibly" A:Act $]\!]$ t b s io = final A t b s io ;

(17) final $[\![$ "unfolding" U:Unf $]\!]$ t b s io = unf-final U $[\![$ "unfolding" U $]\!]$ t b s io;

(18) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) final A' t b s' io' = completed t'' b'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act $[\![$ "and" "then" $]\!]$ A':Act $]\!]$ t b s io =
completed (t', t'') b'' s'' io'' either(c', c'') ;

(19) (1) final A t b s io = completed t' empty-map s' io' c';

(2) final A' t b s' io' = escaped t'' b'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act $[\![$ "and" "then" $]\!]$ A':Act $]\!]$ t b s io = escaped t'' s'' io'' either(c', c'') ;

(20) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) final A' t b s' io' = failed s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act $[\![$ "and" "then" $]\!]$ A':Act $]\!]$ t b s io = failed s'' io'' either(c', c'') ;

(21) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) final A' t b s' io' = completed t'' b'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act "then" A':Act $]\!]$ t b s io = completed t'' b'' s'' io'' either(c', c'') ;

(22) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) final A' t' b' s' io' = escaped t'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act "then" A':Act $]\!]$ t b s io = escaped t'' s'' io'' either(c', c'') ;

(23) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) final A' t' b s' io' = failed s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act "then" A':Act $]\!]$ t b s io = failed s'' io'' either(c', c'') ;

(24) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t overlay(b', b) s' io' = completed t'' b'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act "before" A':Act $]\!]$ t b s io =

completed (t', t'') overlay(b'', b) s'' io'' either(c', c'') ;

(25) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t overlay(b', b) s' io' = escaped t'' s'' io'' c''
$\Rightarrow$ final $[\![$ A:Act "before" A':Act $]\!]$ t b s io = escaped t'' s'' io'' either(c', c'') ;

(26) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t' overlay(b', b) s' io' = failed s'' io'' c''
$\Rightarrow$ final ⟦ A:Act "before" A':Act ⟧ t b s io = failed s'' io'' either(c', c'') ;

(27) (1) final A t b s io = escaped t' s' io' c' ;

(2) final A' t' b s' io' = completed t'' b'' s'' io'' c''
$\Rightarrow$ final ⟦ A:Act "trap" A':Act ⟧ t b s io = completed t'' b'' s'' io'' either(c', c'') ;

(28) (1) final A t b s io = escaped t' s' io' c' ;

(2) final A' t' b s' io' = escaped t'' s'' io'' c''
$\Rightarrow$ final ⟦ A:Act "trap" A':Act ⟧ t b s io = escaped t'' s'' io'' either(c', c'') ;

(29) (1) final A t b s io = escaped t' s' io' c' ;

(2) final A' t' b s' io' = failed s'' io'' c''
$\Rightarrow$ final ⟦ A:Act "trap" A':Act ⟧ t b s io = failed s'' io'' either(c', c'') ;

(30) (1) final A t b s io = failed s' io' uncommitted
$\Rightarrow$ final ⟦ A:Act "or" A':Act ⟧ t b s io = final A' t s io ;

(31) (1) final A t b s io = failed s' io' committed
$\Rightarrow$ final ⟦ A:Act "or" A':Act ⟧ t b s io = failed s' io' committed ;

(32) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t' overlay(b', b) s' io' = completed t'' empty-map s'' io'' c''

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "hence" A':Act ⟧ t b s io =
completed (t', t'') empty-map s'' io'' either(c', c'') ;

(33) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t' overlay(b', b) s' io' = escaped t'' s'' io'' c''

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "hence" A':Act ⟧ t b s io =
escaped t'' s'' either(c', c'') ;

(34) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t' overlay(b', b) s' io' = failed s'' io'' c''

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "hence" A':Act ⟧ t b s io =
failed s'' either(c', c'') ;

(35) (1) final A t b s io = completed t' b' s' io' c' ;

(2) final A' t' overlay(b', b) s' io' = completed t'' empty-map s'' io'' c''

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "thence" A':Act ⟧ t b s io =

completed t'' empty-map s'' io'' either(c', c'') ;

(36) (1) final $A\ t\ b\ s\ io$ = completed $t'\ b'\ s'\ io'\ c'$;

(2) final $A'\ t'$ overlay(b', b) $s'\ io'$ = escaped $t''\ s''\ io''\ c''$

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "thence" A':Act ⟧ $t\ b\ s\ io$ =
escaped $t''\ s''$ either(c', c'') ;

(37) (1) final $A\ t\ b\ s\ io$ = completed $t'\ b'\ s'\ io'\ c'$;

(2) final $A'\ t'$ overlay(b', b) $s'\ io'$ = failed $s''\ io''\ c''$

$\Rightarrow$ final ⟦ ⟦ "furthermore" A:Act ⟧ "thence" A':Act ⟧ $t\ b\ s\ io$ =
failed $s''\ io''$ either(c', c'' ;

(38) (1) final $A\ t\ b\ s\ io = m_a$

$\Rightarrow$

(2) (1) m_a : escaped | failed

$\Rightarrow$

(2) final ⟦ A:Act O:(⟦ "and" "then" ⟧ | "then" | "before") A':Act ⟧
$t\ b\ s\ io = m_a$;

(3) final ⟦ ⟦ "furthermore" A:Act ⟧ O:("hence" | "thence") A':Act ⟧
$t\ b\ s\ io = m_a$;

(3) (1) m_a : completed | failed
$\Rightarrow$ final ⟦ A: Act "trap" A':Act ⟧ $t\ b\ s\ io = m_a$;

(4) (1) m_a : completed | escaped
$\Rightarrow$ final ⟦ A: Act "or" A':Act ⟧ $t\ b\ s\ io = m_a$;

(39) (1) multi-evaluated $D'\ t\ b\ s$ = nothing
$\Rightarrow$ final ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ $t\ b\ s\ io$ =
failed $s\ io$ uncommitted ;

(40) (1) evaluated $D\ t\ b\ s$ = nothing

$\Rightarrow$

(2) final ⟦ "give" D:Dependent ⟧ $t\ b\ s\ io$ = failed $s\ io$ uncommitted ;

(3) final ⟦ "check" D:Dependent ⟧ $t\ b\ s\ io$ = failed $s\ io$ uncommitted ;

(4) final ⟦ "bind" k:token "to" D:Dependent ⟧ $t\ b\ s\ io$ = failed $s\ io$ uncommitted ;

(5) final ⟦ "store" D:Dependent "in" D':Dependent ⟧ $t\ b\ s\ io$ =

failed s io uncommitted ;

(6) final ⟦ "store" D':Dependent "in" D:Dependent ⟧ t b s io =
failed s io uncommitted ;

(7) final ⟦ "batch-send" D:Dependent ⟧ t b s io = failed s io uncommitted ;
failed s io uncommitted ;

(8) ⟦ "enact" "application" D:Dependent ⟧ t b s io = failed s io uncommitted ;
failed s io uncommitted ;

A.3.2 Unfolding

needs: Actions .

introduces: unf-final _ _ _ _ _ _ .

- unfinal _ _ _ _ _ _ :
  Unf, ⟦ "unfolding" Unf ⟧, data, bindings, storage, input-output $\rightarrow$ state .

  t, t', t'' : data ;
  b, b'' : bindings ;
  s, s', s'' : storage ;
  io, io', io'' : input-output ;
  m_a : state ;
  c', c'' : commitment

$\Rightarrow$

(1) (1) final A t b s io = completed () empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t b s' io' = completed t'' b'' s'' io'' c'' ;

(3) O: ⟦ "and" "then" ⟧ | "before"

$\Rightarrow$ unf-final ⟦ A: Act O U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
completed t'' b'' s'' io'' either(c', c'') ;

(2) (1) final A t b s io = completed () empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t b s' io' = escaped t'' s'' io'' c'';

(3) O: ⟦ "and" "then" ⟧ | "before"

$\Rightarrow$ unf-final ⟦ A: Act O U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =

escaped t'' s'' io'' either(c', c'') ;

(3) (1) final A t b s io = completed () empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t b s' io' = failed s'' io'' c'';

(3) O: ⟦ "and" "then" ⟧ | "before"

⇒ unf-final ⟦ A: Act O U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
failed s'' io'' either(c', c'') ;

(4) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = completed t'' b'' s'' io'' c''

⇒ unf-final ⟦ A: Act "then" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
completed t'' b'' s'' io'' either(c', c'') ;

(5) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = escaped t'' s'' io'' c''

⇒ unf-final ⟦ A: Act "then" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
escaped t'' s'' io'' either(c', c'') ;

(6) (1) final A t b s io = completed t' empty-map s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = failed s'' io'' c''

⇒ unf-final ⟦ A: Act "then" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
failed s'' io'' either(c', c'') ;

(7) (1) final A t b s io = escaped t' s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = completed t'' b'' s'' io'' c''

⇒ unf-final ⟦ A: Act "trap" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
completed t'' b'' s'' io'' either(c', c'') ;

(8) (1) final A t b s io = escaped t' s' io' c'

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = escaped t'' s'' io'' c'';

⇒ unf-final ⟦ A: Act "trap" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
escaped t'' s'' io'' either(c', c'') ;

(9) (1) final A t b s io = escaped t' s' io' c' ;

(2) unf-final U ⟦ "unfolding" U' ⟧ t' b s' io' = failed s'' io'' c''

⇒ unf-final ⟦ A: Act "trap" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =

failed s'' io'' either(c', c'') ;

(10) (1) final A t b s io = failed s' io' uncomitted

$\Rightarrow$ unf-final ⟦ A: Act "or" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
unf-final U ⟦ "unfolding" U' ⟧ t b s io ;

(11) (1) final A t b s io = failed s' io' committed

$\Rightarrow$ unf-final ⟦ A: Act "or" U: Unf ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =
failed s' io' committed ;

(12) unf-final ⟦ U: Unf "or" A: Act ⟧ ⟦ "unfolding" U' : Unf ⟧ t b s io =

unf-final ⟦ A "or" U ⟧ ⟦ "unfolding" U' ⟧ t b s io ;

(13) unf-final "unfold" ⟦ "unfolding" U: Unf ⟧ t b s io = final ⟦ "unfoldings" ⟧ t b s io ;

(14) (1) final A t b s io = m_a

$\Rightarrow$

(2) (1) m_a : escaped | failed
$\Rightarrow$ unf-final ⟦ A: Act O: (⟦ "and" "then" ⟧ | "then" | "before") U: Unf ⟧
⟦ "unfolding" U': Unf ⟧ t b s io = m_a ;

(3) (1) m_a completed | failed
$\Rightarrow$ unf-final ⟦ A: Act "trap" U: Unf ⟧ ⟦ "unfolding" U': Unf ⟧ t b s io = m_a ;

(4) (1) m_a completed | escaped
$\Rightarrow$ unf-final ⟦ A: Act "or" U: Unf ⟧ ⟦ "unfolding" U': Unf ⟧ t b s io = m_a .

A.3.3 Tuples

needs: Dependent Data .

introduces: multi-evaluated _ _ _ _ .

- multi-evaluated _ _ _ _ :: Tuple, data, bindings, storage → data (*partial* .

  t : data ;
  b : bindings ;
  s : storage ;

$\Rightarrow$

(1) multi-evaluated "()" $t\ b\ s = $ () ;

(2) multi-evaluated D:Dependent $t\ b\ s =$ evaluated $D\ t\ b\ s$;

(3) multi-evaluated ⟦ D: Tuple "," D' Tuple ⟧ $t\ b\ s =$
multi-evaluated $D\ t\ b\ s$, multi-evaluated $D'\ t\ b\ s$;

(4) multi-evaluated "them" $t\ b\ s = t$.

A.3.4 Dependent Data

needs: Unary Operations , Binary Operations , Data .

introduces: evaluated _ _ _ _ .

- evaluated _ _ _ _ :: Dependent, data, bindings, storage → datum (*partial* .

  t : data ;
  b : bindings ;
  s : storage ;

⇒

(1) evaluated "true" $t\ b\ s =$ true ;

(2) evaluated "false" $t\ b\ s =$ false ;

(3) evaluated n: natural $t\ b\ s = n$

(4) evaluated ⟦ "empty-list" "&" "[" T:Type "]" "list" ⟧ $t\ b\ s =$ empty-list ;

(5) evaluated ⟦ "closure" "abstraction" "of" A: Act "&" "[" "perhaps" "using"
D: Data "]" "act" ⟧ $t\ b\ s =$ closure-abstraction $A\ D\ b$;

(6) evaluated ⟦ O: Unary D: Dependent ⟧ $t\ b\ s =$ unary-operation O (evaluated $D\ t\ b\ s$) ;

(7) evaluated ⟦ O: Binary "(" D: Dependent "," D': Dependent ")" ⟧ $t\ b\ s =$
binary-operation O: (evaluated $D\ t\ b\ s$) (evaluated $D'\ t\ b\ s$;

(8) evaluated ⟦ D: Dependent O:("is" | ⟦ "is" "less" "than" ⟧) D': Dependent⟧ $t\ b\ s =$
binary-operation O: (evaluated $D\ t\ b\ s$) (evaluated $D'\ t\ b\ s$) ;

(9) evaluated ⟦ "Component#" O: Dependent "items" D': Dependent ⟧ $t\ b\ s =$
binary-operation "at" (evaluated $D\ t\ b\ s$) (evaluated $D'\ t\ b\ s$) ;

(10) evaluated "it" $t\ b\ s =$ t & datum ;

(11) evaluated $[\![$ “the” “given” D: Datum “#” n: natural $]\!]$ $t\ b\ s =$
(datum D) & (component# $n\ t$) ;

(12) evaluated $[\![$ “the” $D =$ Datum “bound” “to” k:token $]\!]$ $t\ b\ s =$ (datum D) & (b at k) ;

(13) evaluated $[\![$ “the” D: Datum “stored” “in” D': Dependent $]\!]$
$t\ b$ (m: storage-map, n: natural) $=$

(datum D) & (m at ((evaluated $D\ t\ b\ (m, n)$) & cell)) ;

(14) evaluated $[\![$ “(” $D =$ Dependent ‘)” $]\!]$ $t\ b\ s =$ evaluated $D\ t\ b\ s$.

A.3.5 Unary Operations

introduces: unary-operation _ _ .

- unary-operation _ _ :: Unary, datum $\rightarrow$ datum (*partial*).

  v : datum ;

$\Rightarrow$

(1) unary-operation “not” $v =$ not v ;

(2) unary-operation “negation” $v =$ negation v ;

(3) unary-operation $[\![$ “list” “of” $]\!]$ $v =$ list of v :

(4) unary-operation “head” $v =$ head of v

(5) unary-operation “tail” $v =$ tail of v

A.3.6 Binary Operations

introduces: binary-operation _ _ _ .

- binary-operation _ _ _ ::
  Binary | “is” | $[\![$ ‘is” “less” “than” $]\!]$ | “at”, datum, datum $\rightarrow$ datum (*partial*) .

  v, v': datum ;

$\Rightarrow$

(1) binary-operation “both” $v\ v' =$ both (v, v') ;

(2) binary-operation “either” $v\ v'$ = either $(v,\ v')$;

(3) binary-operation “sum” $v\ v'$ = sum$(v,\ v')$;

(4) binary-operation “difference” $v\ v'$ = difference$(v,\ v')$;

(5) binary-operation “concatenation” $v\ v'$ = concatenation$(v,\ v')$;

(6) binary-operation “is” $v\ v'$ = v is v' ;

(7) binary-operation ⟦ “is” “less” “than” ⟧ $v\ v'$ = v is less than v' ;

(8) binary-operation “at” $v\ v'$ = component # v (items v') .

A.3.7 Data

introduces: datum _, data _, small-datum _ .

- datum _ :: Datum → datum .
- data _ :: Data → data .
- small-datum _ :: Type → datum .

(1) datum “datum” = datum .

(2) datum “cell” = cell .

(3) datum “abstraction” = abstraction .

(4) datum “list” = [datum] list .

(5) datum ⟦ S': “Datum” “|” S'' = Datum ⟧ = (datum S') | (datum S'') .

(6) datum T : Type = small-datum T .

(7) data () = () .

(8) data T : Type = small-datum T .

(9) data ⟦ D : Data “,” D' : Data ⟧ = (data D, data D') .

(10) small-datum “truth-value” = truth-value .

(11) small-datum “integer” = integer .

(12) small-datum ⟦ “truth-value” “cell”⟧ = truth-value-cell .

(13) small-datum ⟦ “integer” “cell”⟧ = integer-cell .

(14) small-datum ⟦ “[” T: Type “]” “list”⟧ = [small-datum T] list .

Appendix B

A Pseudo SPARC Mashine Language

B.1 Abstract Syntax

needs: Data Notation/Numbers/Integers .

introduces: instruction , movable , argument , register , page-id ,
skip , call , return , storeregisters , loadregisters ,
jump _ , branchequal _ , branchlessthan _,
store _ in _ _ _ , load _ _ _ into _ , move _ to _ , compare _ with _ ,
sum _ _, difference _ _ ,
global-register , firstfree , sp , hp , cp , cef , global , arg ,
return-address , staticlink , general-register , reg _ ,
stack , store , heap , commits , input , output .

(1) skip , call , return , storeregisters , loadregisters : instruction .

(2) jump _ , branchequal _ , branchlessthan _ :: integer $\rightarrow$ instruction $(total)$.

(3) store _ in _ _ _ :: register, register, integer, page-id $\rightarrow$ instruction $(total)$.

(4) load _ _ _ into _ :: register, integer, page-id, register $\rightarrow$ instruction $(total)$.

(5) move _ to _ :: movable, register $\rightarrow$ instruction $(total)$.

(6) compare _ with _ :: register, argument $\rightarrow$ instruction $(total)$.

(7) argument $\leq$ movable .

(8) sum _ _ , difference _ _ :: register, argument → movable (*total*) .

(9) argument = register | integer .

(10) register = global-register | staticlink | general-register .

(11) global-register = firstfree | sp | hp | cp | cef | global | arg (*individual*) .

(12) reg _ :: natural → general-register (*total*) .

(13) page-id = stack | store | heap | commits | input | output (*individual*) .

B.2 Semantic Entities

includes: Data Notation .

needs: Abstract Syntax .

introduces: program , linenumber , program-counter , was-zero , was-negative , globals , windows , registers , memory , page , update _ _ , sparc-state , program _ , program-counter _ , globals _ , windows _ , memory _ , finished _ _ _ , out-of-bound _ , _ is in _ , _ at _ default _ .

(1) program = [linenumber to instruction] map .

(2) linenumber = natural .

(3) program-counter = linenumber .

(4) was-zero = truth-value .

(5) was-negative = truth-value .

(6) globals = [global-registers to integer] map .

(7) windows = [registers$^+$] list .

(8) registers = [return-address | staticlink | general-register to integer] map .

(9) memory = [page-id to page] map .

(10) page = [natural to integer] map .

(11) update _ _ :: windows, registers → windows (*total*) .

(12) update w:windows x:registers = concatenation(list of overlay(x, head of w), tail of w) .

(13) sparc-state =

(program, program-counter, was-zero, was-negative, globals, windows, memory) .

(14) program _ :: sparc-state $\rightarrow$ program $(total)$.

(15) program-counter _ :: sparc-state $\rightarrow$ program-counter $(total)$.

(16) globals _ :: sparc-state $\rightarrow$ globals $(total)$.

(17) windows _ :: sparc-state $\rightarrow$ windows $(total)$.

(18) memory _ :: sparc-state $\rightarrow$ memory $(total)$.

(19) program m_p:sparc-state = component#1 of m_p .

(20) program-counter m_p:sparc-state = component#2 of m_p .

(21) globals m_p:sparc-state = component#5 of m_p .

(22) windows m_p:sparc-state = component#6 of m_p .

(23) memory m_p:sparc-state = component#7 of m_p .

(24) finished _ _ _ :: sparc-state, linenumber*, natural $\rightarrow$ truth-value $(total)$.

(25) finished m_p:sparc-state lt:linenumber* nw:natural =
both((program-mounter of m_p) is in lt, (count of items of windows of m_p) is nw) .

(26) out-of-bound _ :: sparc-state $\rightarrow$ truth-value $(total)$.

(27) out-of-bound m_p:sparc-state =
(maximum of mapped-set of program of m_p) is less than (program-counter of m_p) .

(28) _ is in _ :: natural, natural* $\rightarrow$ truth-value $(total)$.

(29) n: natural is in () = false .

(30) n: natural is in n': natural = n is n' .

(31) n: natural is in (nt: natural*,nt': natural*) = either(n is in nt, n is in nt') .

(32) _ at _ default _ :: $[X$ to $Y]$ map , x: X, y: $Y \rightarrow Y$ $(total)$.

(33) (M: $[X$ to $Y]$ map) at x: X default y: Y =
if X is in mapped-set of M then M at x else y .

B.3 Semantic Functions

needs: Abstract Syntax , Semantic Entities .

B.3.1 Programs

needs: Instructions .

introduces: sparc-final _ _ _ _ , step _ .

- sparc-final _ _ _ _ :: natural, sparc-state, linenumber*, natural → sparc-state ($partial$) .

- step _ :: sparc-state → sparc-state ($total$) .

(1) sparc-final n:natural m_p:sparc-state il:linenumber* nw:natural =

```
    if both($n$ is 0, finished $m_p$ $lt$ $nw$)
    then $m_p$,
    else if not any( $n$ is 0, finished $m_p$ , $lt$ $nw$, out-of-bound $m_p$)
        then sparc-final predecessor($n$) step($m_p$) $lt$ $nw$
        else nothing .
```

(2) step m_p:sparc-state =
next ((program of m_p) at (program-counter of m_p) default skip) m_p .

B.3.2 Instructions

needs: Moveables , Arguments , Registers .

introduces: next _ _ .

- next _ _ :: instruction , sparc-state → sparc-state (*total*) .
  p : program ;
  pc : program-counter ;
  cz : was-zero ;
  cn : was-negative ;
  g : globals ;
  w : windows ;
  q : memory

⇒

(1) next skip (p, pc, cz, cn, g, w, q) = (p, sum(pc, 1), cz, cn, g, w, q);

(2) next call (p, pc, cz, cn, g, w, q) =
(p, g at global default 0, cz, cn, g, update w (map of return-address to pc), q) ;

(3) next return (p, pc, cz, cn, g, w, q) =
(p, sum((head of w) at return-address default 0, 1), cz, cn, g, w, q) ;

(4) next storeregisters (p, pc, cz, cn, g, w, q) =
    (p, sum(pc,1), cz, cn, g, concatenation(list of empty-map, w), q) ;

(5) next loadregisters (p, pc, cz, cn, g, w, q) = (p, sum(pc,1), cz, cn, g,
    (if (tail of w) is (empty-list) then w else (tail of w)), q) ;

(6) next (jump i:integer) (p, pc, cz, cn, g, w, q) = (p, i, cz, cn, g, w, q) ;

(7) next (branchequal i:integer) (p, pc, true, cn, g, w, q) = (p, i, true, cn, g, w, q) ;

(8) next (branchequal i:integer) (p, pc, false, cn, g, w, q) =
    (p, sum(pc, 1), false, cn, g, w q) ;

(9) next (branchlessthan i:integer) (p, pc, cz, true, g, w, q) = (p, i, cz, true, g, w, q) ;

(10) next (branchlessthan i:integer) (p, pc, cz, false, g, w, q) =
    (p, sum(pc, 1), cz false g, w q) ;

(11) next (store R':register in R'':register i:integer P:page-id) (p, pc, cz, cn, g, w, q) =
    (p, sum(pc, 1), cz, cn, g, w,
    overlay(map of p to overlay(map of sum(fetch R'' g w, i) to
    fetch R' g w, q at P default empty-map), q)) ;

(12) (1) (q at P default empty-map) at sum(fetch R g w, i) default 0 = i':integer
$\Rightarrow$
(2) next (load R:register i:integer P:page-id into G:global-register)
    (p, pc, cz, cn, g, w, q) =
    (p, sum(pc, 1), cz, cn, overlay(map of G to i', g), w, q) ;

(3) next (load R:register i:integer P:page-id into x:(staticlink | general-register))
    (p, pc, cz, cn, g, w, q) =
    (p, sum(pc, 1), cz, cn, g, update w (map of x to i'), q) ;

(13) (1) evaluate-movable M g w = i:integer
$\Rightarrow$
(2) next (move M:movable to G:global-register)
    (p, pc, cz, cn, g, w, q) =
    (p, sum(pc, 1), cz, cn, overlay(map of G to i, g), w, q) ;

(3) next (move M:movable to x:(staticlink | general-register))
    (p, pc, cz, cn, g, w, q) =
    (p, sum(pc, 1), cz, cn, g, update w (map of x to i), q) ;

(14) (1) fetch R g w = i':integer ;

(2) evaluate-argument $A\ g\ w = i'$:integer
$\Rightarrow$ next (compare R:register with A:argument) (p, pc, cz, cn, g, w, q) =
(p, sum(pc, 1), i' is i'', i' is less than i'', g, w, q) .

B.3.3 Moveables

needs: Arguments , Registers .

introduces: evaluate-movable _ _ _ .
- evaluate-movable _ _ _ :: movable, globals, windows $\rightarrow$ integer ($total$) .
  g : globals ;
  w : windows

$\Rightarrow$

(1) evaluate-movable A:argument $g\ w$ = evaluate-argument $A\ g\ w$;

(2) (1) fetch $R\ g\ w = i'$:integer ;

(2) evaluate-argument $A\ g\ w = i''$:integer
$\Rightarrow$
(3) evaluate-movable (sum R:register A:argument) $g\ w$ = sum(i', i'') ;

(4) evaluate-movable (difference R:register A:argument) $g\ w$ = difference(i', i'') .

B.3.4 Arguments

needs: Registers .

introduces: evaluate-argument _ _ _ .
- evaluate-argument _ _ _ :: argument, globals, windows $\rightarrow$ integer ($total$) .
  g : globals ;
  w : windows

$\Rightarrow$

(1) evaluate-argument R:register $g\ w$ = fetch $R\ g\ w$;

(2) evauate-argument i:integer $g\ w = i$.

B.3.5 Registers

introduces: fetch _ _ _ .
- fetch _ _ _ :: register, globals, windows $\rightarrow$ integer ($total$) .

g : globals ;
w : windows

$\Rightarrow$

(1) fetch G:global-register $g\ w = g$ at G default 0 ;

(2) fetch staticlink $g\ w =$ (head of w) at staticlink default 0 ;

(3) fetch r:general-register $g\ w =$ (head of w) at r default 0 .

Appendix C

Actions to SPARC Compiler

needs: **Data Notation ,**
A Compilable Subset of Action Notation/Abstract Syntax ,
A Pseudo SPARC Machine Language/Abstract Syntax .

C.1 Compile Time Entities

C.1.1 Types

needs: **Data Types , Symbol Tables .**

introduces: type , truth-value-type , integer-type , cell-type ,
truth-value-cell-type , integer-cell-type , abstraction-type ,
list-type , abstraction-type _ _ _ _ _ _ , list-type _ , storable-type _ .

(1) type = truth-value-type | integer-type | cell-type | abstraction-type | list-type (*disjoint*) .

(2) cell-type = truth-value-cell-type | integer-cell-type (*individual*) .

(3) truth-value-type , integer-type : type .

(4) abstraction-type _ _ _ _ _ _ :: data-type, truth-value, data-type,
truth-value, data-type, symbol-table → abstraction-type (*total*) .

(5) list-type _ :: type → list-type (*total*) .

(6) _ is _ :: type, type → truth-value (*total, commutative*) .

(7) truth-value-type is S:(integer-type | cell-type | abstaction-type | list-type) = false .

(8) integer-type is S:(cell-type | abstraction-type | list-type) = false .

(9) truth-value-cell-type is S:(integer-cell-type | abstraction-type | list-type) = false .

(10) integer-cell-type is S:(abstraction-type | list-type) = false .

(11) (1) h, h', h_n, h'_n, h_e, h'_e : data-type ;

(2) z_n , z'_n , z_e , z'_e : truth-value ;

(3) d , d' : symbol-table

⇒ (abstraction-type h z_n h_n z_e h_e d) is (abstraction-type h' z'_n h'_n z'_e h'_e d') =
all(h is h', either(not(z_n is z'_n), h_n is h'_n),
either(not(z_e is z'_e), h_e is h'_e), d is d') .

(12) S:abstraction-type is S':list-type = false .

(13) (list-type S:type) is (list-type S':type) = S is S .

(14) storable-type _ :: cell-type → truth-value-type | integer-type (*total*) .

(15) storable-type truth-value-cell-type = truth-value-type .

(16) storable-type integer-cell-type = integer-type .

C.1.2 Data Types

needs: Types .

introduces: data-type , abstraction-free _ , compare-data-types _ _ _ _ .

h , h' : data-type

⇒

(1) data-type = type* ;

(2) abstraction-free _ :: type* → truth-value (*total*) ;

(3) abstraction-free () = true ;

(4) abstraction-free S:abstraction-type = false ;

(5) abstraction-free S(truth-value-type | integer-type | cell-type | list-type) = true ;

(6) abstraction-free S:type , S':type) = both(abstraction-free S, abstraction-free S') ;

(7) compare-data-types _ _ _ _ ::
truth-value, data-type, truth-value, data-type → data-type (*partial*) ;

(8) compare-data-types z h z' h' =
if z' is false then h else
if both(z is false, z' is true) then h' else

if all(z is true, z' is true, h is h') then h else nothing .

C.1.3 Registers

introduces: frozen , _ is less than _ , successor _ , minimum of _ , maximum of _ , free-register _ , registers up to _ .

(1) frozen = [general-register] set .

(2) _ is less than _ :: general-register, general-register → truth-value (*total*) .

(3) (reg n:natural) is less than (reg n':natural) = n is less than n' .

(4) successor _ :: general-register → general-register (*total*) .

(5) successor reg n:natural = reg successor n .

(6) minimum of _ :: set → natural (*partial*) .
set of element$^+$ → natural(*partial*) .

(7) maximum of _ :: set → natural (*partial*) .
set of element$^+$ → natural(*partial*) ;

(8) minimum of empty-set = nothing .

(9) minimum of set of x:element = x .

(10) minimum of union(x:set, y:set) = if (minimum of x) is
less than (minimum of y) then minimum of x else minimum of y .

(11) maximum of empty-set = nothing .

(12) maximum of set of x:element = x .

(13) maximum of union(x:set, y:set) = if (maximum of x) is
less than (maximum of y) then maximum of y else maximum of x .

(14) free-register _ :: frozen → general-register (*total*) .

(15) free-register f:frozen = if f is empty-set then reg 0 else
minimum of difference(registers up to successor(maximum of f), f) .

(16) registers up to _ :: general-register → set of general-registers$^+$ (*total*) .

(17) registers up to (reg 0) = set of (reg 0) .

(18) registers up to (successor r:general-register) =
union(registers up to r, successor(r)) .

C.1.4 Symbol Tables

needs: Types .

introduces: symbol-table , block , entry , entry _ _ ,
cleanup , compare-blocks _ _ _ _ , insert _ _ .

d : symbol-table ;
e, e' : block

$\Rightarrow$

(1) symbol-table = [block] list ;

(2) block = [entry] list ;

(3) entry _ _ :: token, type $\rightarrow$ entry ($total$) ;

(4) cleanup = natural ;

(5) compare-blocks _ _ _ _ :: truth-value, block, truth-value, block $\rightarrow$ block ($total$) ;

(6) compare-blocks z e z' e' =
if z' is false then e else
if both(z is false, z' is true) then e' else
if all(z is true, z' is true, e is e') then e else nothing .

(7) insert _ _ :: symbol-table, block $\rightarrow$ symbol-table ($total$) ;

(8) insert d e = concatenation(list of(concatenation(e, head of d)), tail of d) .

C.1.5 Miscellaneous

needs: Data Types , Symbol Tables .

introduces: ac-state , ac-state _ _ _ _ _ _ , a-state , a-state _ _ _ ,
wc-state , wc-state _ _ , code-size _ ,
linenumber-complete , linenumber-escape , linenumber-fail ,
linenumber-unfold , error .

(1) ac-state _ _ _ _ _ _ :: natural, truth-value, data-type,
truth-value, data-type, block $\rightarrow$ ac-state ($total$) .

(2) a-state _ _ _ :: program, general-register, general-register $\rightarrow$ a-state ($total$) .

(3) wc-state _ _ :: natural, data-type $\rightarrow$ wc-state ($total$) .

(4) code-size _ : : wc-state $\rightarrow$ natural ($total$) .

(5) code-size (wc-state n: natural h:data-type() $= n$.
(6) linenumber-complete = linenumber .
(7) linenumber-escape = linenumber .
(8) linenumber-fail = linenumber .
(9) linenumber-unfold = linenumber .
(10) error : error .
(11) error is x:tuple = false .
(12) error is x:wc-state = false .

C.2 Code Macros

needs: Compile Time Entities/Miscellaneous .

C.2.1 Lists and Tuples

introduces: empty-list-code _ _ , single-list-code _ _ _ , concatenation-code _ _ _ _ ,
head-code _ _ _ _ , tail-code _ _ _ _ , at-code _ _ _ _ _ ,
e-size , s-size , c-size , h-size , t-size , a-size .

- empty-list-code _ _ :: general-register, linenumber $\rightarrow$ program ($total$) .
- single-list-code _ _ _ :: general-register, general-register, linenumber $\rightarrow$ program ($total$) .
- concatenation-code _ _ _ _ ::
  general-register, general-register, general-register, linenumber $\rightarrow$ program ($total$) .
- head-code _ _ _ _ ::
  general-register, general-register, linenumber, linenumber $\rightarrow$ program ($total$) .
- tail-code _ _ _ _ ::
  general-register, general-register, linenumber, linenumber $\rightarrow$ program ($total$) .
- at-code _ _ _ _ _ :: general-register, general-register, general-register,
  linenumber, linenumber $\rightarrow$ program ($total$) .

r , r' , r'' : general-register ;
l : linenumber ;
l_f : linenumber-fail

$\Rightarrow$

```
(1)   empty-list-code $r$ $l$ = overlay(
          map of sum($l$,0) to ( move -1 to global ) ,
          map of sum($l$,1) to ( store global in hp 1 heap ) ,
          map of sum($l$,2) to ( move hp to $r$ ) ,
          map of sum($l$,3) to ( move sum hp 2 to hp )) ;
(2)   single-list-code $r$ $r'$ $l$ = overlay(
          map of sum($l$,0) to ( store $r$ in hp 2 heap ) ,
          map of sum($l$,1) to ( move -1 to global ) ,
          map of sum($l$,2) to ( store global in hp 1 heap ) ,
          map of sum($l$,3) to ( store hp in hp 3 heap ) ,
          map of sum($l$,4) to ( move sum hp 2 to $r'$ ) ,
          map of sum($l$,5) to ( move sum hp 4 to hp )) ;
(3)   concatenation-code $r$ $r'$ $r''$ $l$ = overlay(
          map of sum($l$,0) to ( load $r$ 1 heap into global ) ,
          map of sum($l$,1) to ( compare global with -1 ) ,
          map of sum($l$,2) to ( branchequal sum($l$,4) ) ,
          map of sum($l$,3) to ( jump sum($l$,6) ) ,
          map of sum($l$,4) to ( move $r$ to $r''$ ) ,
          map of sum($l$,5) to ( jump sum($l$,18) ) ,
          map of sum($l$,6) to ( move hp to $r''$ ) ,
          map of sum($l$,7) to ( move $r$ to global ) ,
          map of sum($l$,8) to ( load global 0 heap into arg ) ,
          map of sum($l$,9) to ( store arg in hp 0 heap ) ,
          map of sum($l$,10) to ( load global 1 heap into global ) ,
          map of sum($l$,11) to ( load global 1 heap into arg ) ,
          map of sum($l$,12) to ( move sum hp 2 to hp ) ,
          map of sum($l$,13) to ( store hp in hp -1 heap ) ,
          map of sum($l$,14) to ( compare arg with -1 ) ,
          map of sum($l$,15) to ( branchequal sum($l$,17) ) ,
          map of sum($l$,16) to ( jump sum($l$,8) ) ,
          map of sum($l$,17) to ( store $r'$ in hp -1 heap )) ;
(4)   head-code $r$ $r'$ $l$ $l_f$ = overlay(
          map of sum($l$,0) to ( load $r$ 1 heap into global ) ,
          map of sum($l$,1) to ( compare global with -1 ) ,
          map of sum($l$,2) to ( branchequal $l_f$ ) ,
          map of sum($l$,3) to ( load $r$ 0 heap into $r'$ )) ;
```

(5) tail-code r r' l l_f = overlay(
map of sum(l,0) to (load r 1 heap into r') ,
map of sum(l,1) to (compare r' with -1) ,
map of sum(l,2) to (branchequal l_f) ,

(6) at-code r r' r'' l l_f= overlay(
map of sum(l,0) to (compare r' with 1) ,
map of sum(l,1) to (branchlessthan l_f) ,
map of sum(l,2) to (move r to global) ,
map of sum(l,3) to (move r' to arg) ,
map of sum(l,4) to (compare arg with 1) ,
map of sum(l,5) to (branchequal sum(l,11)) ,
map of sum(l,6) to (load global 1 heap into global) ,
map of sum(l,7) to (compare global with -1) ,
map of sum(l,8) to (branchequal l_f) ,
map of sum(l,9) to (move difference arg 1 to arg) ,
map of sum(l,10) to (jump sum(l,4)) ,
map of sum(l,11) to (load global 0 heap into r'') ,

(7) e-size = 4 ;

(8) s-size = 6 ;

(9) c-size = 18 ;

(10) h-size = 4 ;

(11) t-size = 3 ;

(12) a-size = 12 .

C.2.2 Committing

introduces: putcommit _ _ , combinecommit _ , combine _ _ _ _ .

- putcommit _ _ :: linenumber, 0 | 1 → program ($total$) .
- combinecommit _ :: linenumber → program ($total$) .
- combine _ _ _ _ :: linenumber,
  linenumber-complete, linenumber-escape, linenumber-fail → program ($total$) .

i : 0 | 1 ;
l : linenumber ;

l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail

$\Rightarrow$

(1) putcommit l i = overlay(
map of sum(l,0) to (move i to global) ,
map of sum(l,1) to (store global in cp 0 commits) ,
map of sum(l,2) to (move sum cp 1 to cp) ,

(2) combinecommit l = overlay(
map of sum(l,0) to (move difference cp 1 to cp) ,
map of sum(l,1) to (load cp 0 commits into global) ,
map of sum(l,2) to (compare global with 0) ,
map of sum(l,3) to (branchequal sum(l,5)) ,
map of sum(l,4) to (store global in cp -1 commits)) ,

(3) combine l l_n l_e l_f = overlay(
map of sum(l,5) to (jump l_n) ,
combinecommit sum(l,6) ,
map of sum(l,11) to (jump l_e) ,
combinecommit sum(l,12) ,
map of sum(l,17) to (jump l_f) ,

C.2.3 Bookkeeping

introduces: finalize _ _ _ _ .

• finalize _ _ _ _ :: linenumber, cleanup 0 | 1 | 2, linenumber → program (*total*) .
l, l' : linenumber-complete
u : cleanup
i : 0 | 1 | 2

$\Rightarrow$

(1) finalize l u i l' = overlay(
map of sum(l,0) to (move difference sp u to sp) ,
map of sum(l,1) to (move i to cef) ,
map of sum(l,2) to (jump l')) .

C.2.4 Call

introduces: call-sequence _ _ _ _ _ _ _ _ _ , return-sequence _ _ _ .

- call-sequence _ _ _ _ _ _ _ _ _ ::
  linenumber, general-register, general-register, cleanup, cleanup, cleanup,
  linenumber-complete, linenumber-escape, linenumber-fail $\rightarrow$ program $(total)$.
- return-sequence _ _ _ ::
  general-register, general-register, linenumber $\rightarrow$ $(total)$.

l: linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_{ef} : linenumber-fail ;
r , r' , a_n , a_e : general-register ;
u_n , u_e , u_f : cleanup

$\Rightarrow$

(1) call-sequence l r r' u_n u_e u_f l_n l_e l_f = overlay(
map of sum(l,0) to (move sum sp 1 to sp) ,
map of sum(l,1) to (load r 1 heap into global) ,
map of sum(l,2) to (store global in sp 0 stack)) ,
map of sum(l,3) to (load r 0 heap into global) ,
map of sum(l,4) to (move r' to arg) ,
map of sum(l,5) to (storeregisters) ,
map of sum(l,6) to (move sp to staticlink) ,
map of sum(l,7) to (move arg to reg 0) ,
map of sum(l,8) to (call) ,
map of sum(l,9) to (loadregisters) ,
map of sum(l,10) to (move arg to r) ,
map of sum(l,11) to (compare cef with 0) .
map of sum(l,12) to (branchequal sum(l, 17)) ,
map of sum(l,13) to (compare cef with 1) ,
map of sum(l,14) to (branchequal sum(l,19)) .
map of sum(l,15) to (move difference sp sum(u_f, 1) to sp) ,
map of sum(l,16) to (jump l_f) ,
map of sum(l,17) to (move difference sp sum(u_n, 1) tp sp) .
map of sum(l,18) to (jump l_n) ,
map of sum(l,19) to (move difference sp sumu_e, 1) tp sp) ,
map of sum(l,20) to (jump l_e)) .

putcommit sum(l,21) 0 ,
finalize sum(l,24) u_f 2 l_f) ,

(2) return-sequence a_n a_e l = overlay(
map of sum(l,0) to (move a_n to arg) ,
map of sum(l,1) to (jump sum(l, 3)) ,
map of sum(l,2) to (move a_e to arg) ,
map of sum(l,3) to (return)) .

C.3 Analysis

needs: **Compile Time Entities .**

C.3.1 Actions

needs: **Unfolding , Tuples , Dependent Data .**

introduces: a-count _ _ _ .

- a-count _ _ _ :: Act, data-type, symbol-table → ac-state (*partial*) .

h, h'_n, h''_n, h'_e, h''_e : data-type ;
d : symbol-table ;
l_e : linenumber-escape ;
z'_n , z''_n , z'_e z''_e : truth-value ;
e , e' , e'' : block
n' , n'' : natural

⇒

(1) a-count "complete" h d = ac-state sum(e-size,6) true () false () empty-list ;

(2) a-count "escape" h d = ac-state 6 false () true h empty-list ;

(3) a-count "fail" h d = ac-state 6 false () false () empty-list ;

(4) a-count "commit" h d = ac-state sum(e-size,6) true () false () empty-list ;

(5) a-count "diverge" h d = ac-state 1 false () false () empty-list ;

(6) a-count "regive" h d =
ac-state 6 true h false () empty-list ;

(7) (1) d-count $D\ h\ d =$ (n:natural, S:type)
$\Rightarrow$ a-count ⟦ "give" D:Dependent ⟧ $h\ d =$
ac-state sum(n,s-size,12) true S false () emptylist ;

(8) (1) d-count $D\ h\ d =$ (n:natural, truth-value-type)
$\Rightarrow$ a-count ⟦ "check" D:Dependent ⟧ $h\ d =$
ac-state sum(n,2,e-size,12) true () false () emptylist ;

(9) (1) d-count $D\ h\ d =$ (n:natural, S:type)
$\Rightarrow$ a-count ⟦ "bind" k:token "to" D:Dependent ⟧ $h\ d =$
ac-state sum(n,2,e-size,11) true () false () (list of entry $k\ S$) ;

(10) (1) d-count $D\ h\ d =$ (n':natural, S':(storable-type S'')
(2) d-count $D'\ h\ d =$ (n'':natural, S'':(cell-type)
$\Rightarrow$ a-count ⟦ "store" D:Dependent "in" D':Dependent ⟧ $h\ d =$
ac-state sum(n', n'',3,e-size,12) true () false () emptylist ;

(11) a-count ⟦ "allocate" "truth-value" "cell" ⟧ $h\ d =$
ac-state sum(1,s-size,9) true truth-value-cell-type false () empty-list ;

(12) a-count ⟦ "allocate" "integer" "cell" ⟧ $h\ d =$
ac-state sum(1,s-size,9) true integer-cell-type false () empty-list ;

(13) (1) d-count $D\ h\ d =$ (n:natural, integer-type)
$\Rightarrow$ a-count ⟦ "batch-send" D:Dependent ⟧ $h\ d =$
ac-state sum(n,5,e-size,12) true () false () emptylist ;

(14) a-count ⟦ "batch-receive" "an" "integer" ⟧ $h\ d =$
ac-state sum(5,s-size,9) true integer false () empty-list ;

(15) (1) d-count $D\ h\ d =$ (n:natural,
abstraction-type h':data-type z_n:truth-value h_n:data-type
z_e:truth-value h_e:data-type d':symbol-table) ;
(2) w-count $D'\ h\ d =$ wc-state n':natural h':data-type ;
$\Rightarrow$ a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ $h\ d =$
ac-state sum(n,n',27) $z_n\ h_n\ z_e\ h_e$ emptylist ;

(16) a-count ⟦ "indivisibly" A:Act ⟧ $h\ d =$ a-count $A\ h\ d$;

(17) (1) u-count $U\ h\ d$ false () false () empty-list = ac-state $n\ z_n\ h_n\ z_e\ h_e\ e$
$\Rightarrow$ a-count ⟦ "unfolding" U:Unf ⟧ $h\ d =$ ac-state sum (4,n,18) $z_n\ h_n\ z_e\ h_e\ e$;

(18) (1) a-count $A\ h\ d =$ ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e$ empty-list ;
(2) a-count $A'\ h\ d =$ ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e$;
(3) compare-data-types $z'_e\ h'_e\ z''_e\ h''_e = h_e$:data-type
$\Rightarrow$ a-count ⟦ A:Act ⟦"and" "then"⟧ A':Act ⟧ $h\ d =$ ac-state sum(n',2,n'',c-size, 18)

both(z_n', z_n'') (h_n', h_n'') either(z_e', z_e'') h_e e ;

(19) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' empty-list ;
(2) a-count A' h_n' d = ac-state n'' z_n'' h_n'' z_e'' h_e'' e ;
(3) compare-data-types z_e' h_e' z_e'' h_e'' = h_e:data-type
$\Rightarrow$ a-count $[\![$ A:Act "then" A':Act $]\!]$ h d = ac-state sum(n',2,n'', 18)
both(z_n', z_n'') h_n'' either(z_e', z_e'') h_e e ;

(20) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' e' ;
(2) a-count A' h insert (d, e') = ac-state n'' z_n'' h_n'' z_e'' h_e'' e'' ;
(3) compare-data-types z_e' h_e' z_e'' h_e'' = h_e:data-type
$\Rightarrow$ a-count $[\![$ A:Act "before" A':Act $]\!]$ h d = ac-state sum(n',2,n'',c-size, 18)
both(z_n', z_n'') (h_n', h_n'') either(z_e', z_e'') h_e concatenation(e'',, e');

(21) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' e' ;
(2) a-count A' h_e' d = ac-state n'' z_n'' h_n'' z_e'' h_e'' e'' ;
(3) compare-data-types z_n' h_n' z_n'' h_n'' = h_n:data-type
(4) compare-blocks z_n' e' z_n'' e'' = e:block
$\Rightarrow$ a-count $[\![$ A:Act "trap" A':Act $]\!]$ = ac-state sum(n',2,n'',18)
either(z_n',z_n'') $h_n$both(z_e',z_e'') h_e'' e ;

(22) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' e' ;
(2) a-count A' h d = ac-state n'' z_n'' h_n'' z_e'' h_e'' ;
(3) compare-data-types z_n' h_n' z_n'' h_n'' = h_n:data-type ;
(4) compare-data-types z_e' h_e' z_e'' h_e'' = h_e:data-type ;
(5) compare-blocks z_n' e' z_n'' e'' = e:block ;
$\Rightarrow$ a-count $[\![$ A:Act "or" A':Act $]\!]$ =
ac-state sum(n',7,n'',18)
either(z_n',z_n'') (z_e',z_e'') h_e e ;

(23) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' e' ;
(2) a-count A' h insert(d,e') = ac-state n'' z_n'' h_n'' z_e'' h_e'' empty-list ;
(3) compare-data-types z_e' h_e' z_e'' h_e'' = h_e:data-type ;
(4) abstraction-free (h_n' h_e' h_n'' h_e'') = true
$\Rightarrow$ a-count $[\![$ $[\![$ "furthermore" A':Act $]\!]$ "hence" A': Act $]\!]$ h d =
ac-state sum(n',2,n'',c-size,18) both(z_n',z_n'') (h_n',h_n'')
either(z_e',z_e'') h_e empty-list ;

(24) (1) a-count A h d = ac-state n' z_n' h_n' z_e' h_e' e' ;
(2) a-count A' h_n' insert(d,e') = ac-state n'' z_n'' h_n'' z_e'' h_e'' empty-list ;
(3) compare-data-types z_e' h_e' z_e'' h_e'' = h_e:data-type ;
(4) abstraction-free (h_e' h_n'' h_e'') = true

$\Rightarrow$ a-count ⟦ ⟦ "furthermore" A':Act ⟧ "hence" A': Act ⟧ h d =
  ac-state sum(n',2,n'',18) both(z'_n,z''_n) h''_n either(z'_e,z''_e) h_e emptylist :

(25) (1) w-count D' h d = error
$\Rightarrow$ a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ h d =
  ac-state 6 false() false () empty-list ;

(26) (1) d-count D h d = error
$\Rightarrow$
(2) a-count ⟦ "give" D:Dependent ⟧ h d = ac-state 6 false () false () empty-list ;
(3) a-count ⟦ "check" D:Dependent ⟧ h d = ac-state 6 false () false () empty-list ;
(4) a-count ⟦ "bind" k:token "to" D:Dependent ⟧ h d =
  ac-state 6 false () false () empty-list ;
(5) a-count ⟦ "store" D:Dependent "in" D':Dependent ⟧ h d =
  ac-state 6 false () false () empty-list ;
(6) a-count ⟦ "store" D':Dependent "in" D:Dependent ⟧ h d =
  ac-state 6 false () false () empty-list ;
(7) a-count ⟦ "batch-send" D:Dependent ⟧ h d =
  ac-state 6 false () false () empty-list ;
(8) a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ h d =
  ac-state 6 false () false () empty-list ;

C.3.2 Unfolding

needs: **Actions .**

introduces: u-count _ _ _ _ _ _ _ _ _ .

• u-count _ _ _ _ _ _ _ _ :: Unf, data-type, symbol-table, truth-value,
  data-type, truth-value, data-type, block → ac-state (*partial*) .

h, h_n, h'_n, h''_n, h_e, h'_e, h''_e : data-type ;
d : symbol-table ;
z_n, z'_n, z''_n, z_e, z'_e z''_e : truth-value ;
e , e' , e'' : block
n' , n'' : natural

$\Rightarrow$
(1) (1) a-count A h d = ac-state n' z'_n () z'_e h'_e empty-list ;

(2) compare-data-types z_e h_e z'_e h'_e = h'''_e:data-type ;
(3) u-count U h d z_n h_n either(z_e,z'_e) h'''_e e = ac-state n'' z''_n h''_n z''_e h''_e e''
⇒ u-count ⟦ A:Act O:(⟦ "and" "then" ⟧ | "before") U:Unf ⟧ h d z_n h_n z_e h_e e
= ac-state sum(n',7,n'') z''_n h''_n z''_e h''_e e'' ;

(2) (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e empty-list ;
(2) compare-data-types z_e h_e z'_e h'_e = h'''_e:data-type ;
(3) u-count U h'_n d z_n h_n either(z_e,z'_e) h'''_e e = ac-state n'' z''_n h''_n z''_e h''_e e''
⇒ u-count ⟦ A:Act "then" U:Unf ⟧ h d z_n h_n z_e h_e e =
ac-state sum(n',7,n'') z''_n h''_n z''_e h''_e e'' ;

(3) (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e e' ;
(2) compare-data-types z_n h_n z'_n h'_n = h'''_n:data-type ;
(3) compare-blocks z_n e z'_n e' = e''':block ;
(4) u-count U h'_e d either(z_n,z'_n) h'''_n z_e h_e e''' = ac-state n'' z''_n h''_n z''_e h''_e e''
⇒ u-count ⟦ A:Act "trap" U:Unf ⟧ h d z_n h_n z_e h_e e =
ac-state sum(n',7,n'') z''_n h''_n z''_e h''_e e'' ;

(4) (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e e' ;
(2) compare-data-types z_n h_n z'_n h'_n = h'''_n:data-type ;
(3) compare-data-types z_e h_e z'_e h'_e = h'''_e:data-type ;
(4) compare-blocks z_n e z'_n e' = e''':block ;
(5) u-count U h d either(z_n,z'_n) h'''_n either(z_e,z'_e) h'''_e e''' =
ac-state n'' z''_n h''_n z''_e h''_e e''
⇒ u-count ⟦ A:Act "or" U:Unf ⟧ h d z_n h_n z_e h_e e =
ac-state sum (n',12,n'') z''_n h''_n z''_e h''_e e'' ;

(5) u-count ⟦ U:Unf "or" A:Act ⟧ h d z_n h_n z_e h_e e =
u-count ⟦ A "or" U ⟧ h d z_n h_n z_e h_e e ;

(6) u-count "unfold" h d z_n h_n z_e h_e e = ac-state 2 z_n h_n z_e h_e e .

C.3.3 Tuples

needs: **Dependent Data .**

introduces: w-count _ _ _

- w-count _ _ _ ::
  Tuples, data-type, symbol-table → (natural, data-type) | error (*partial*) .

h : data-type ;

d : symbol-table ;

⇒

(1) w-count "()" h d = wc-state e-size () ;

(2) (1) d-count D h d = (n:natural, S:type)
⇒ w-count D:Dependent h d = wc-state sum(n,s-size) S ;

(3) (1) d-count D h d = error
⇒ w-count D:Dependent h d = error ;

(4) (1) w-count T h d = wc-state n:natural h':datatype ;
(2) w-count T' h d = wc-state n:natural h'':datatype
⇒ w-count ⟦T:Tuple "," T':Tuple ⟧ h d = wc-state sum(n,n',c-size) (h',h'') ;

(5) (1) d-count D h d = error
⇒
(2) w-count ⟦T:Tuple "," T':Tuple ⟧ h d = error ;
(3) w-count ⟦T':Tuple "," T':Tuple ⟧ h d = error ;

(6) w-count "them" h d = wc-state 0 h .

C.3.4 Dependent Data

needs: **Actions , Lookup in Symbol Tables,
Unary Operations , Binary Operations , Data .**

introduces: d-count _ _ _ .

• d-count _ _ _ ::
Dependent, data-type, symbol-table → (natural, type) | error (*partial*) .

h , h' , h_n , h_e : data-type ;
d : symbol-table ;
z_n , z_e : truth-value ;
n : natural

⇒

(1) d-count "true" h d = (1, truth-value-type) ;

(2) d-count "false" h d = (1, truth-value-type) ;

(3) d-count n:natural h d = (1, integer-type) ;

(4) d-count ⟦ "empty-list" "&" "[" T:Type "]" "list" ⟧ h d = (e-size, list-type, type T) ;

(5) (1) data-type $D = h'$;
(2) a-count A h' concatenation(list of empty-list d) = ac-state z_n h_n z_e h_e empty-list
$\Rightarrow$ d-count ⟦ “closure” “abstraction” “of” A:Act “&” “[” “perhaps” “using” D:Data “]” “act” ⟧ h d =
(sum(n, 10), abstraction-type h' z_n h_n z_e h_e d) ;

(6) (1) data-type D h d = (n:natural, S:type);
(2) unary-count O S = (n':natural, S:type)
$\Rightarrow$ d-count ⟦ O:Unary D:Dependent ⟧ h d = (sum(n, n'), s') ;

(7) (1) data-type D h d = (n:natural, S:type);
(2) unary-count O S = error
$\Rightarrow$ d-count ⟦ O:Unary D:Dependent ⟧ h d = error ;

(8) (1) d-count D h d = = error
$\Rightarrow$ d-count ⟦ O:Unary D:Dependent ⟧ h d = error ;

(9) (1) d-count D h d = (n:natural, S:type) ;
(2) d-count D' h d = (n'natural, S':type) ;
(3) binary-count O S S' =(n'':natural, S'':type)
$\Rightarrow$
(4) d-count ⟦ O:Binary “(” D:Dependent “,” D':Dependent “)” ⟧ h d =
(sum(n,n',n''), S'') ;
(5) d-count ⟦ D:Dependent O:(“is” | ⟦ “is” “less” “than” ⟧) D':Dependent ⟧
h d = (sum(n,n',n''), S'') ;

(10) (1) d-count D h d = (n:natural, S:type) ;
(2) d-count D' h d = (n'natural, S':type) ;
(3) binary-count O S S' =error
$\Rightarrow$
(4) d-count ⟦ O:Binary “(” D:Dependent “,” D':Dependent “)” ⟧ h d = error ;
(5) d-count ⟦ D:Dependent O:(“is” | ⟦ “is” “less” “than” ⟧) D':Dependent ⟧
h d = error ;

(11) (1) d-count D h d = error
$\Rightarrow$
(2) d-count ⟦ D:Dependent O:(“is” | ⟦ “is” “less” “than” ⟧) D':Dependent ⟧
h d = error ;
(3) d-count ⟦ D':Dependent O:(“is” | ⟦ “is” “less” “than” ⟧) D:Dependent ⟧
h d = error ;
(4) d-count ⟦ O:Binary “(” D:Dependent “,” D':Dependent “)” ⟧ h d = error ;

(5) d-count $[\![$ O:Binary "(" D':Dependent "," D:Dependent ")" $]\!]$ h d = error ;
(6) d-count $[\![$ "component#" D:Dependent "items" D':Dependent ")" $]\!]$ h d = error ;
(7) d-count $[\![$ "component#" D':Dependent "items" D:Dependent ")" $]\!]$ h d = error ;

(12) (1) d-count D h d = (n:natural, S:type) ;
(2) d-count D' h d = (n'natural, S':type) ;
(3) binary-count "at" S S' =(n'':natural, S'':type)
$\Rightarrow$ d-count $[\![$"component#" D':Dependent "items" D:Dependent $]\!]$ h d =
(sum(n, n' n''), S'') ;

(13) (1) d-count D h d = (n:natural, S:type) ;
(2) d-count D' h d = (n':natural, S':type) ;
(3) binary-count "at" S S' = error
$\Rightarrow$ d-count $[\![$"component#" D':Dependent "items" D:Dependent $]\!]$ h d = error ;

(14) (1) h:type) ;
$\Rightarrow$ d-count "it" h d = (sum(1,a-size,h) ;

(15) (1) h & type = nothing ;
$\Rightarrow$ d-count "it" h d = error ;

(16) (1) component # n of h) : (datum-type S)
$\Rightarrow$ d-count $[\![$"the" "given" S:Datum "#" n:natural $]\!]$ h d =
(sum(1, a-size), component #n of h) ;

(17) (1) (component # n of h) & (datum-type S) = nothing
$\Rightarrow$ d-count $[\![$"the" "given" S:Datum "#" n:natural $]\!]$ h d = error ;

(18) (1) find-count k d = (n:natural, S':type, j:natural) ;
(2) S' : (datum-type S)
$\Rightarrow$ d-count $[\![$"the" S:Datum "bound" "to" k:token $]\!]$ h d = (sum(n,2),s') ;

(19) (1) find-count k d = (n:natural, S':type, j:natural) ;
(2) S' & (datum-type S) = nothing
$\Rightarrow$ d-count $[\![$"the" S:Datum "bound" "to" k:token $]\!]$ h d = error ;

(20) (1) find-count k d = error
$\Rightarrow$ d-count $[\![$"the" S:Datum "bound" "to" k:token $]\!]$ h d = error ;

(21) (1) d-count D h d = (n:natural, S':cell-type) ;
(2) storable-type S' = S ;
(3) S' : (datum-type S)
$\Rightarrow$ d-count $[\![$"the" S:Datum "stored" "in" D:Dependent $]\!]$ h d = (sum(n,4), S'') ;

(22) (1) d-count D h d = (n:natural, S':cell-type) ;
(2) storable-type S' = S'' ;

(3) S'' & (datum-type S) = nothing
$\Rightarrow$ d-count $[\![$ "the" S:Datum "stored" "in" D:Dependent $]\!]$ h d = error ;

(23) (1) d-count D h d = (n:natural, S':cell-type) ;
(2) S' & cell-type = nothing
$\Rightarrow$ d-count $[\![$ "the" S:Datum "stored" "in" D:Dependent $]\!]$ h d = error ;

(24) (1) d-count D h d = error ;
$\Rightarrow$ d-count $[\![$ "the" S:Datum "stored" "in" D:Dependent $]\!]$ h d = error ;

(25) d-count $[\![$ "(" D:Dependent ")" $]\!]$ h d = d-count D h d .

C.3.5 Lookup in Symbol Tables

introduces: find-count _ _ , block-find-count _ _ .

- find-count _ _ :: token, symbol-table → (natural, type, natural) | error (*total*) .
- block-find-count _ _ :: token, block → (type, natural) | error (*total*) .

(1) find-count k:token empty-list = error .

(2) (1) block-find–count k e = (S:type, j:natural)
$\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table) =(0 S j) .

(3) (1) block-find–count k e = error ;
(2) find–count k d = (n:natural, S:type, j:natural)
$\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table)
(sum(n, 1), S, j) .

(4) (1) block-find–count k e = error ;
(2) find–count k d = error
$\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table) = error .

(5) block-find–count k:token empty-list = error .

(6) block-find-count k:token concatenation(list of entry k':token S:type, e:block) =
if k is k'
then (S, sum(count of items of e,1))
else block-find-count k e .

C.3.6 Unary Operations

introduces: unary-count _ _

- unary-count _ _ :: Unary, type → (natural, type) | error (*total*) .

(1) unary-count “not” S:type =
if S is truth-value-type then (2, truth-value-type) else error .

(2) unary-count “negation” S:type = if S is integer-type then (2, integer-type) else error .

(3) unary-count ⟦ “list” “of” ⟧ S:type = (s-size, list-type S) .

(4) unary-count “head” (list-type S:type) = (h-size, S) .

(5) unary-count “tail” S:list-type = (t-size, S) .

(6) (1) S & list-type = nothing
⇒
(2) unary-count “head” S:type = error ;
(3) unary-count “tail” S:type = error .

C.3.7 Binary Operations

introduces: binary-count _ _ _

- binary-count _ _ _ :: Binary | “is” | ⟦ “is” “less” “than” ⟧ | “at”, type, type →
(natural, type) | error (*total*) .

(1) binary-count O:(both” | “either”) S:type S':type =
if both(S is S', S is truth-value-type) then (7, truth-value-type) else error .

(2) binary-count O: (”sum | “difference”) S:type S':type =
if both(S is S', S is integer-type) then (1, integer-type) else error .

(3) binary-count “concatenation” S:list-type S':list-type =
if S is S', S then (c-size, S) else nothing .

(4) (1) S & list-type = nothing
⇒
(2) binary-count “concatenation” S:type S':type = error ;
(3) binary-count “concatenation” S':type S:type = error .

(5) binary-count “is” S:type S':type =
if both(S is S', either(S is truth-value-type, S is integer-type,

S is truth-value-cell-type, S is integer-cell-type))
then (5, truth-value-type) else error .

(6) binary-count ⟦"is" "less" "than"⟧ S:type S':type =
if both(S is S', S is integer-type) then (5, truth-value-type) else error .

(7) binary-count "at" (list-type S:type) S':type =
if S' is integer-type then (a-size, S) else error .

(8) (1) S & list-type = nothing
$\Rightarrow$ binary-count "at" S:type S':type = error ;

C.3.8 Data

introduces: datum-type _, data-type _, type _ .

- datum-type _ :: Datum $\rightarrow$ type .
- data-type _ :: Data $\rightarrow$ data-type ($total$) .
- type _ :: Type $\rightarrow$ type ($total$) .

(1) datum-type "datum" = type .

(2) datum-type "cell" = cell-type .

(3) datum-type "abstraction" = abstraction-type .

(4) datum-type "list" = list-type .

(5) datum-type ⟦ S:Datum "|" S':Datum ⟧ = (datum-type S) | (datum-type S') .

(6) datum-type T:Type = type T .

(7) data-type "()" = () .

(8) data-type T:Type = type T .

(9) data-type ⟦ D:Data "," D':Data ⟧ = (data-type D, data-type D') .

(10) type "truth-value" = truth-vale-type .

(11) type "integer" = integer-type .

(12) type ⟦ "truth-value" "cell" ⟧ = truth-vale-cell-type .

(13) type ⟦ "integer" "cell" ⟧ = integer-cell-type .

(14) type ⟦ "[" T:Type "]" "list" ⟧ = list-type (type T) .

C.4 Code Generation

needs: **Compile Time Entities , Code Macros , Analysis .**

C.4.1 Actions

needs: **Unfolding , Thples , Dependent Data .**

introduces: perform _ _ _ _ _ _ _ _ _ _ _ _ .

• perform _ _ _ _ _ _ _ _ _ _ _ _ ::
Act, data-type, general-register, frozen, symbol-table,
cleanup, cleanup, cleanup, linenumber, linenumber-complete,
linenumber-escape, linenumber-fail $\rightarrow$ a-state ($partial$) .

h , h_n , h'_n , h''_n , h_e , h'_e , h''_e : data-type ;
a , a' , a_n , a'_n , a''_n , a_e , a'_e , a''_e , r , r' : general-register ;
f , f' : frozen ;
d : symbol-table ;
u_n , u_e , u_f : cleanup ;
l , l' , l'', l''' : linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
j , n , n' , n'' : natural ;
z_n , z'_n , z''_n , z_e , z'_e , z''_e : truth-value ;
e , e' , e'' : block ;
p , p' , p'' : program

$\Rightarrow$

(1) (1) $r =$ free-register f
$\Rightarrow$ perform "complete" $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f =$ a-state overlay(
empty-list-code $r\ l$, putcommit sum(l,3) 0, finalize sum(l,6) u_n 0 l_n) $r\ a$;

(2) perform "escape" $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f =$ a-state overlay(
putcommit l 0, finalize sum(l,3) u_e 1 l_e) $a\ a$;

(3) perform "fail" $h\ a\ f\ d\ h\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f =$ a-state overlay(
putcommit l 0, finalize sum(l,3) u_f 2 l_f) $a\ a$;

(4) (1) $r =$ free-register f

```
        ⇒    perform "complete" $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state overlay(
                      empty-list-code $r\ l$ , putcommit sum($l$,3) 1, finalize sum($l$,6) $u_n$ 0 $l_n$) $r\ a$ ;
(5)     perform "diverge" $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state (map of $l$ to (jump $l$ )) $a\ a$ ;
(6)     perform "regive" $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state overlay(
                      putcommit $l$ 0, finalize sum($l$,3) $u_n$ 2 $l_n$) $a\ a$ ;
(7)     (1)  d-count $D\ h\ d = (n$:natural, $S$:type) ;
        (2)  $l' =$ sum$(l, n)$ ;
        (3)  $l'' =$ sum($l'$,s-size) ;
        (4)  evaluate $D\ h\ a\ f\ d\ l$ sum($l''$,6) = ($p$:program, $r$:general-register) ;
        (5)  $r' =$ free-register union($f$,set of $r$) ;
        ⇒
        (6)  perform ⟦ "give" $D$:Dependent ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state overlay(
                   $p$ ,
                   single-list-code $r\ r'\ l'$ ,
                   putcommit $l''$ 0 ,
                   finalize sum($l''$,3) $u_n$ 0 $l_n$ ,
                   putcommit sum($l''$,6) 0 ,
                   finalize sum($l''$,9) $u_f$ 2 $l_f$ )

                   $r'\ a$ ;
(8)     (1)  d-count $D\ h\ d = (n$:natural, truth-value-type) ;
        (2)  $l' =$ sum$(l, n)$ ;
        (3)  $l'' =$ sum($l'$,2,e-size) ;
        (4)  evaluate $D\ h\ a\ f\ d\ l$ sum($l''$,6) = ($p$:program, $r$:general-register) ;
        ⇒    perform ⟦ "check" $D$:Dependent ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state overlay(
                   $p$ ,
                   map of sum($l'$,0) to ( compare $r$ with 0 ) ,
                   map of sum($l'$,1) to ( branchequal sum($l''$,6) ) ,
                   empty-list-code $r$  sum($l'$,2) ,
                   putcommit $l''$0 ,
                   finalize sum($l''$,3) $u_n$ 0 $l_n$
                   putcommit sum($l''$,6) 0 ,
                   finalize sum($l''$,9) $u_f$ 2 $l_f$ )

                   $r\ a$ ;
(9)     (1)  d-count $D\ h\ d = (n$:natural, $S$:type) ;
        (2)  $l' =$ sum$(l, n)$ ;
```

```
(3)     $l'' = \text{sum}(l',2,\text{e-size})$ ;
(4)     evaluate $D\ h\ a\ f\ d\ l$ sum($l''$,5) = ($p$:program, $r$:general-register) ;
(5)     $u_n$ is 0 = true
$\Rightarrow$   perform ⟦ "bind" $k$:token "to" $D$:Dependent ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
            a-state overlay(
            $p$ ,
            map of sum($l'$,0) to ( move sum sp 1 to sp ),
            map of sum($l'$,1) to ( store $r$ in sp 0 stack ) ,
            empty-list-code $r$ sum($l'$,2) ,
            putcommit $l''$ 0 ,
            putcommit $l''$ 0 ,
            map of sum($l''$,3) to ( move 0 to cef ) ,
            map of sum($l''$,4) to ( jump $l_n$ ) ,
            putcommit sum($l''$,5) 0 ,
            finalize sum($l''$,8) $u_f$ 2 $l_f$ )

            $r\ a$ ;
(10)  (1)     d-count $D\ h\ d$ = ($n'$:natural, $S'$:type) ;
      (2)     d-count $D'\ h\ d$ = ($n''$:natural, $S''$:type) ;
      (3)     $l' = \text{sum}(l, n')$ ;
      (4)     $l'' = \text{sum}(l', n'')$ ;
      (5)     $l''' = \text{sum}(l',3,\text{e-size})$ ;
      (6)     evaluate $D\ h\ a$ union($f$, set of $a$) $d\ l$ sum($l'''$,6) =
                  ($p$:program, $r$:general-register) ;
      (7)     $f'$ = union($f$, set of $r'$) ;
      (8)     evaluate $D'\ h\ a\ f'\ d\ l'$ sum($l'''$,6) = ($p''$:program, $r''$:general-register)
      $\Rightarrow$   perform ⟦ "store" $D$:Dependent "in" $D'$:Dependent ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
            a-state overlay(
            $p'$ ,
            $p''$ ,
            map of sum($l''$,0) to ( move 1 to global ),
            map of sum($l''$,1) to ( store global in $r''$ 0 store ) ,
            map of sum($l''$,2) to ( store $r'$ in $r''$ 1 store )) ,
            empty-list-code $r'$ sum($l''$,3) ,
            putcommit $l'''$ 1 ),
            finalize sum( $l'''$ ,3) $u_n$ 0 $l_n$ ) ,
            putcommit sum($l'''$,6) 0 ),
        finalize sum($l'''$,9) $u_f$ 2 $l_f$ )

        $r'$ $a$ ;
(11)  (1)  $r$ = free-register $f$
      (2)  $r'$ = free-register union($f$, set of $r$) ;
      (3)  $l'$ = sum($l$,1,s-size)
      ⇒    perform ⟦ “allocate” $x$:(“truth-value” | “integer”) “cell” ⟧
              $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state overlay(
              map of sum($l$,0) to ( move firstfree to $r$ ),
              single-list-code $r$ $r'$ sum($l$,1) ,
              map of $l'$ to ( move 0 to global ) ,
              map of sum($l'$,1) to ( store global in $r$ 0 store )) ,
              map of sum($l'$,2) to ( move sum fristfree 2 to firstfree )) ,
              putcommit sum($l'$,3) 1 ,
              finalize sum($l'''$,6) $u_n$ 0 $l_n$ ))

              $r'$ $a$ ;
(12)  (1)  d-count $D\ h\ d$ = ($n$:natural,integer-type) ;
      (2)  $l'$ = sum($l,n$) ;
      (3)  $l''$ = sum($l'$,5,e-size) ;
      (4)  evaluate $D\ h\ a\ f\ d\ l$ sum($l''$,6) = ($p$:program, $r$:general-register) ;
      (5)  $r'$ = free-register uinon($f$, set of $r$)
      ⇒    perform ⟦ “batch-send” $D$:Dependent ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
              a-state overlay(
              $p$ ,
              map of sum($l'$,0) to ( move 0 to global ),
              map of sum($l'$,1) to ( load global 0 output into arg ) ,
              map of sum($l'$,2) to ( move sum arg 1 to arg ) ,
              map of sum($l'$,3) to ( store arg in global 0 output ) ,
              map of sum($l'$,4) to ( store $r$ in arg 0 output ) ,
              empty-list-code $r'$ sum($l'$,5) ,
              putcommit $l''$ 1 ,
              finalize sum( $l''$ ,3) $u_n$ 0 $l_n$ ,
              putcommit sum($l''$,6) 0 ,
              finalize sum($l''$,9) $u_f$ 2 $l_f$ )

              $r'$ $r'$ ;
```

```
(13)  (1)  $r$ = free-register $f$ ;
      (2)  $r'$ = free-register union($f$, set of $r$) ;
      (3)  $l'$ = sum($l$,7) ;
      (4)  $l''$ = sum($l'$,s-size) ;
      $\Rightarrow$  perform $[\![$ "batch-receive" "an" "integer" $]\!]$ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
              a-state overlay(
              map of sum($l'$,0) to ( move 0 to global ),
              map of sum($l'$,1) to ( load global 0 output into arg ) ,
              map of sum($l'$,2) to ( compare arg with 0 ) ,
              map of sum($l'$,3) to ( branchequal sum($l''$,6) ) ,
              map of sum($l'$,4) to ( load arg 0 input into $r$ ) ,
              map of sum($l'$,5) to ( move difference arg 1 to arg ) ,
              map of sum($l'$,6) to ( store arg into global 0 input ) ,
              single-list-code $r\ r'\ l'$ ,
              putcommit $l''$ 1 ,
              finalize sum( $l''$ ,3) $u_n$ 0 $l_n$ ,
              map of sum($l''$,6) to ( jump sum($l''$,6) ) ,

              $r'\ r'$ ;
(14)  (1)  d-count $D\ h\ d$ = ($n$:natural, abstraction-type $h'$:data-type $z_n$:truth-value
              $h_n$:data-type $z_e$:truth-value $h_e$:data-type $d'$:symbol-table) ;
      (2)  w-count $D'\ h\ d$ = wc-state $n'$natural $h'$data-type ;
      (3)  $l'$ = sum($l, n$) ;
      (4)  $l''$ = sum($l', n'$) ;
      (5)  evaluate $D\ h\ a$ union($f$, set of $a$) $d\ l$ sum($l''$,21) =
              ($p$:program, $r$:general-register) ;
      (6)  with $D'\ h\ a$ union($f$, set of $r$) $d\ l'$ sum($l''$,21) =
              ($p'$:program, $r'$:general-register) ;
      $\Rightarrow$  perform $[\![$ "enact" "application" $D$:Dependent "to" $D'$:Tuple $]\!]$
              $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
              a-state overlay( $p$, $p'$, call-sequence $l''\ r\ r'\ u_n\ u_e\ u_f\ l_n\ l_e\ l_f$) $r\ r$ ;
(15)  perform $[\![$ "indivisibly" $A$:Act $]\!]$ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
              perform $A\ h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ ;
(16)  (1)  u-count $D\ h\ d$ false() false() empty-list = ac-state $n\ z_n\ h_n\ z_e\ h_e\ e$ ;
      (2)  $a'$ = free-register $f$ ;
      (3)  $l'$ = sum($l$,4) ;
      (4)  $l''$ = sum($l', n$) ;
```

(5) unf U h a' union(f, set of a) d u_n u_e u_f a' false() a false() a empty-list
    l' l'' sum(l'',6) sum(l'',12) l' = a-state p a_n a_e ;

(6) either(e is empty-list, u_n is 0) = true

$\Rightarrow$ perform $[\![$ "unfolding" U:Unf $]\!]$ h a f d u_n u_e u_f l l_n l_e l_f = a-state overlay(
    putcommit l 0 ,
    map of sum(l,3) to (move a to a') ,
    p ,
    combine l'' l_n l_e l_f)

    a_n a_e ;

(17) (1) a-count A h d ac-state n' z'_n h'_n z'_e h'_e empty-list ;

(2) a-count A' h d ac-state n'' z''_n h''_n z''_e h''_e e ;

(3) l' = sum(l, n') ;

(4) l'' = sum(l',2,n'') ;

(5) l''' = sum(l'',c-size) ;

(6) perform A h a' union(f, set of a) d 0 u_e u_f
    l sum(l',2) l' l_f = a-state p' a'_n a'_e ;

(7) perform A' h a' union(f, set of a'_n) d u_n u_e u_f
    sum(l',2) l'' sum(l''',6) sum(l''',12) = a-state p'' a''_n a''_e ;

(8) either(e is empty-list, u_n is 0) = true

(9) r = is free-register union(f, set of (a'_n, a''_n))

$\Rightarrow$ perform $[\![$ A:Act $[\![$ "and" "then" $]\!]$ A':Act $]\!]$ h a f d u_n u_e u_f l l_n l_e l_f =
    a-state overlay(
    p'
    map of l' (move a'_e to a''_e) ,
    map of sum(l',1) to (jump l_e) ,
    p'' ,
    concatenation-code a'_n a''_n r l ,
    combine l''' l_n l_e l_f)

    r a''_e ;

(18) (1) a-count A h d ac-state n' z'_n h'_n z'_e h'_e empty-list ;

(2) a-count A' h'_n d ac-state n'' z''_n h''_n z''_e h''_e e ;

(3) l' = sum(l, n') ;

(4) l'' = sum(l',2,n'') ;

(5) perform A h a f d 0 u_e u_f l sum(l',2) l' l_f = a-state p' a'_n a'_e ;

(6) perform A' h'_n a'_n f d u_n u_e u_f sum(l',2) l'' sum(l'',6) sum(l'',12) =

```
        a-state $p'' \; a''_n \; a''_e$ ;
     (7)  either($e$ is empty-list $u_n$ is 0) = true
     ⇒   perform ⟦ $A$:Act "then" $A'$:Act ⟧ $h \; a \; f \; d \; u_n \; u_e \; u_f \; l \; l_n \; l_e \; l_f$ =
        a-state overlay(
        $p'$
        map of $l'$ ( move $a'_e$ to $a''_e$ ) ,
        map of sum($l'$,1) to ( jump $l_e$ ) ,
        $p''$ ,
        combine $l'' \; l_n \; l_e \; l_f$ )

        $a''_n \; a''_e$ ;
(19) (1)  a-count $A \; h \; d$ ac-state $n' \; z'_n \; h'_n \; z'_e \; h'_e \; e'$ ;
     (2)  a-count $A' \; h$ insert($d$,$e'$) = acstate $n'' \; z''_n \; h''_n \; z''_e \; h''_e \; e''$ ;
     (3)  $l'$ = sum($l$,$n'$) ;
     (4)  $l''$ = sum($l'$,2,$n''$) ;
     (5)  $l'''$ = sum($l''$,c-size ) ;
     (6)  perform $A \; h \; a$ union($f$, set of $a$) $d$ 0 $u_e \; u_f$
        $l$ sum($l'$,2) $l' \; l_f$ = a-state $p' \; a'_n \; a'_e$ ;
     (7)  perform $A' \; h \; a$ union($f$, set of $a'_n$) insert($d$,$e'$) $u_n$
        sum($u_e$,count of items of $e'$) sum($u_f$,count of items of $e'$)
        sum($l'$,2) $l''$ sum($l'''$,6) sum($l'''$,12) = a-state $p'' \; a''_n \; a''_e$ ;
     (8)  either(concatenation($e''$,$e'$) is empty-list, $u_n$ is 0) = true ;
     (9)  $r$ = is free-register union($f$, set of ($a'_n$, $a''_n$))
     ⇒   perform ⟦ $A$:Act "before" $A'$:Act ⟧ $h \; a \; f \; d \; u_n \; u_e \; u_f \; l \; l_n \; l_e \; l_f$ =
        a-state overlay(
        $p'$
        map of $l'$ ( move $a'_e$ to $a''_e$ ) ,
        map of sum($l'$,1) to ( jump $l_e$ ) ,
        $p''$ ,
        concatenation-code $a'_n \; a''_n \; r \; l$ ,
        combine $l''' \; l_n \; l_e \; l_f$ )

        $r \; a''_e$ ;
(20) (1)  a-count $A \; h \; d$ = ac-state $n' \; z'_n \; h'_n \; z'_e \; h'_e \; e'$ ;
     (2)  a-count $A' \; h'_e \; d$ = ac-state $n'' \; z''_n \; h''_n \; z''_e \; h''_e \; e''$ ;
     (3)  $l'$ = sum($l, n'$) ;
     (4)  $l''$ = sum($l'$,2,$n''$) ;
```

```
      (5)  perform $A\ h\ a\ f\ d\ u_n$ 0 $l\ l'$ sum($l'$,2) $l_f$ = a-state $p'\ a'_n\ a'_e$ ;
      (6)  perform $A'\ h'_e\ a'_e\ f\ d\ u_n\ u_e\ u_f$ sum($l'$,2) $l''$ sum($l''$,6) sum($l''$,12) =
                   a-state $p''\ a''_n\ a''_e$ ;
      (7)  compare-blocks $z'_n\ e'\ z''_n\ e''$ = $e$:block ;
      (8)  either($e$ is empty-list, $u_n$ is 0) = true
      ⇒    perform ⟦ $A$:Act "trap" $A'$:Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
                   a-state overlay(
                   $p'$
                   map of $l'$ ( move $a'_n$ to $a''_n$ ) ,
                   map of sum($l'$,1) to ( jump $l_n$ ) ,
                   $p''$ ,
                   combine $l''\ l_n\ l_e\ l_f$ )

                   $a''_n\ a''_e$ ;
(21)  (1)  a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e'$ ;
      (2)  a-count $A'\ h\ d$ = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e''$ ;
      (3)  $l'$ = sum($l, n'$) ;
      (4)  $l''$ = sum($l'$,7,$n''$) ;
      (5)  perform $A\ h\ a$ union($f$, set of $a$) $d\ u_n\ u_e$ 0
                   $l\ l'$ sum($l'$,2) sum($l'$,4) = a-state $p'\ a'_n\ a'_e$ ;
      (6)  perform $A'\ h\ a\ f\ d\ u_n\ u_e\ u_f$ sum($l'$,7) $l''$ sum($l''$,6) sum($l''$,12) =
                   a-state $p''\ a''_n\ a''_e$ ;
      (7)  compare-blocks $z'_n\ e'\ z''_n\ e''$ = $e$:block ;
      (8)  either($e$ is empty-list, $u_n$ is 0) = true
      ⇒    perform ⟦ $A$:Act "or" $A'$:Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
                   a-state overlay(
                   $p'$
                   map of $l'$ ( move $a'_n$ to $a''_n$ ) ,
                   map of sum($l'$,1) to ( jump $l_n$ ) ,
                   map of sum($l'$,2) to ( move $a'_e$ to $a''_e$ ) ,
                   map of sum($l'$,3) to ( jump $l_e$ ) ,
                   map of sum($l'$,4) to ( load cp -1 commits into global ),
                   map of sum($l'$,5) to ( compare global with 1 ),
                   map of sum($l'$,6) to ( branchequal $l_f$ ),
                   $p''$
                   combine $l''\ l_n\ l_e\ l_f$ )
```

```
                $a''_n\ a''_e$ ;
(22)    (1)   a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e'$ ;
        (2)   a-count $A'\ h$ insert($d, e'$) = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e$ empty-list ;
        (3)   $l'$ = sum($l, n'$) ;
        (4)   $l''$ = sum($l'$,2,$n''$) ;
        (5)   $l'''$ = sum($l''$, c-size) ;
        (6)   $j$ = count of items of $e$ ;
        (7)   perform $A\ h\ a$ union($f$, set of $a$) $d$ 0 $u_e\ u_f\ l$ sum($l'$,2) $l'\ l_f$ =
                a-state $p'\ a'_n\ a'_e$ ;
        (8)   perform $A'\ h\ a$ union($f$, set of $a'_n$) insert($d, e'$) sum($u_n, j$) sum($u_e, j$) sum($u_f, j$)
                sum($l'$,2) $l''$ sum($l'''$,6) sum($l'''$,12) = a-state $p''\ a''_n\ a''_e$ ;
        (9)   $r$ = free-register union($f$, set of ($a'_n$, $a''_n$))
        ⇒    perform ⟦ ⟦ “furthermore” $A$:Act ⟧ “hence” $A'$:Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
                a-state overlay(
                $p'$ ,
                map of $l'$ to ( move $a'_e$ to $a''_e$ ) ,
                map of sum($l'$,1) to ( jump $l_e$ ) ,
                $p''$
                concatenation-code $a'_n\ a''_n\ r\ l''$ ,
                combine $l'''\ l_n\ l_e\ l_f$ )

                $r\ a''_e$ ;
(23)    (1)   a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e'$ ;
        (2)   a-count $A'\ h'_n$ insert($d$,$e'$) = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e$ empty-list ;
        (3)   $l'$ = sum($l$,$n'$) ;
        (4)   $l''$ = sum($l'$,2,$n''$) ;
        (5)   $j$ = count of items of $e'$ ;
        (6)   perform $A\ h\ a\ d$ 0 $u_e\ u_f\ l$ sum($l'$,2) $l'\ l_f$ =
                a-state $p'\ a'_n\ a'_e$ ;
        (7)   perform $A'\ h'_n\ a'_n\ f$ insert($d$,$e'$) sum($u_n$,$j$) sum($de_e$,$j$) sum($u_f$,$j$)
                sum($l'$,2) $l''$ sum($l''$,6) sum($l''$,12) = a-state $p''\ a''_n\ a''_e$ ;
        ⇒    perform ⟦ ⟦ “furthermore” $A$:Act ⟧ “thence” $A'$:Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ =
                a-state overlay(
                $p'$ ,
                map of $l'$ to ( move $a'_e$ to $a''_e$ ) ,
                map of sum($l'$,1) to ( jump $l_e$ ) ,
                $p''$ ,
```

combine l'' l_n l_e l_f)

a''_n a''_e ;

(24) (1) w-count D' h d = error
⇒ perform ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧
h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;

(25) (1) d-count D h d = error
⇒
(2) perform ⟦ "give" D:Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(3) perform ⟦ "check" D:Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(4) perform ⟦ "bind" k:token "to" D:Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(5) perform ⟦ "store" D:Dependent "in" D':Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(6) perform ⟦ "store" D:Dependent "in" D:Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(7) perform ⟦ "batch-send" D:Dependent ⟧ h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;
(8) perform ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧
h a f d u_n u_e u_f l l_n l_e l_f =
a-state overlay(putcommit l 0, finalize sum(l,3) u_f 2 l_f) a a ;

C.4.2 Unfolding

needs: **Actions .**

introduces: unf _ .

• unf _ ::
Unf, data-type, general-register, frozen, symbol-table,
cleanup, cleanup, cleanup,
general-register, truth-value, general-register,
truth-value, general-register, block
linenumber, linenumber-complete,

linenumber-escape, linenumber-fail, linenumber-unfold → a-state ($partial$) .

h , h_n , h'_n , h''_n , h_e , h'_e , h''_e : data-type ;
a , a' , a_n , a'_n , a''_n , a'''_n , a_e , a'_e , a''_e , a'''_e : general-register ;
f : frozen ;
d : symbol-table ;
u_n , u_e , u_f : cleanup ;
z_n , z'_n , z''_n , z_e , z'_e , z''_e : truth-value ;
e , e' , e'' , e''' : block ;
l , l' : linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
l_u :linenumber-unfold ;
n' , n'' : natural ;
p' , p'' : program

⇒

(1) (1) a-count A h d = ac-state n' z'_n () z'_e h'_e empty-list ;
(2) compare-data-types z_e h_e z'_e h'_e = h'''_e:data-type ;
(3) z_e = either(z_e,z'_e) ;
(4) u-count U h d z_n h_n z'''_e h'''_e e = ac-state n'' z''_n h''_n z''_e h''_e e'' ;
(5) $l' = $ sum(l,n') ;
(6) perform A h a union(f, set of a) d 0 u_e u_f l sum(l',2) l' l_f =a-state p' a'_n a'_e ;
(7) a'''_e = if z_e then a_e else a'_e ;
(8) unf U h a f d u_n u_e u_f a' z_n h_n a_n z'''_e h'''_e a'''_e e sum(l',7) l_n l_e l_f l_u =
        a-state p'' a''_n a''_e ;
(9) either(e'' is empty-list, u_n is 0) = true
⇒ unf ⟦ A:Act O:(⟦ "and" "then" ⟧ "before") U:Unf ⟧
        h a f d u_n u_e u_f a' z_n h_n a_n z_e h_e a_e e l l_n l_e l_f l_u = a-state overlay(
        p ,
        map of l' to (move a'_e to a''_e),
        map of sum(l',1) to (jump l_e),
        combinecommit sum(l',2),
        p'')

        a''_n a''_e ;

(2) (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e empty-list ;
(2) compare-data-types z_e h_e z'_e h'_e = h'''_e:data-type ;

(3) $z_e''' =$ either(z_e,z_e') ;
(4) u-count U h_n' d z_n h_n z_e''' h_e''' $e =$ ac-state n'' z_n'' h_n'' z_e'' h_e'' e'' ;
(5) $l' =$ sum(l, n') ;
(6) perform A h a union(f, set of a) d 0 u_e u_f l sum(l',2) l' l_f =a-state p' a_n' a_e' ;
(7) $a_e''' =$ if z_e then a_e else a_e' ;
(8) unf U h_n' a_n' f d u_n u_e u_f a' z_n h_n a_n z_e''' h_e''' a_e''' e sum(l',7) l_n l_e l_f l_u =
a-state p'' a_n'' a_e'' ;
(9) either(e'' is empty-list, u_n is 0) = true
$\Rightarrow$ unf $[\![$ A:Act "then" U:Unf $]\!]$ h a f d u_n u_e u_f a' z_n h_n a_n z_e h_e a_e e l l_n l_e l_f l_u =
a-state overlay(
p' ,
map of l' to (move a_e' to a_e''),
map of sum(l',1) to (jump l_e),
combinecommit sum(l',2),
p'')

a_n'' a_e'' ;

(3) (1) a-count A h $d =$ ac-state n' z_n' h_n' z_e' h_e' e ;
(2) compare-data-types z_n h_n z_n' $h_n' = h_n'''$:data-type ;
(3) $z_n''' =$ either(z_n, z_n') ;
(4) compare-blocks z_n e z_n' $e' = e'''$:block ;
(5) u-count U h_e' d z_n''' h_n''' z_e h_e $e''' =$ ac-state n'' z_n'' h_n'' z_e'' h_e'' e'' ;
(6) $l' =$ sum(l, n') ;
(7) perform A h a union(f, set of a) d u_n 0 u_f l l' sum(l',2) l_f =a-state p' a_n' a_e' ;
(8) $a_n''' =$ if z_n then z_n else a_n' ;
(9) unf U h_e' a_e' f d u_n u_e u_f a' z_n''' h_n''' a_n''' z_e h_e a_e e''' sum(l',7) l_n l_e l_f l_u =
a-state p'' a_n'' a_e'' ;
(10) either(e'' is empty-list, u_n is 0) = true
$\Rightarrow$ unf $[\![$ A:Act "trap" U:Unf $]\!]$ h a f d u_n u_e u_f a' z_n h_n a_n z_e h_e a_e e l l_n l_e l_f l_u =
a-state overlay(
p' ,
map of l' to (move a_n' to a_n''),
map of sum(l',1) to (jump l_n),
combinecommit sum(l',2),
p'')

a_n'' a_e'' ;

(4) (1) a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e$;
(2) compare-data-types $z_n\ h_n\ z'_n\ h'_n = h'''_n$:data-type ;
(3) z'''_n = either(z_n,z'_n) ;
(4) compare-data-types $z_e\ h_e\ z'_e\ h'_e = h'''_e$:data-type ;
(5) z'''_e = either(z_e,z'_e) ;
(6) compare-blocks $z_n\ e\ z'_n\ e' = e'''$:block ;
(7) u-count $U\ h'_e\ d\ z'''_n\ h'''_n\ z'''_e\ h'''_e\ e'''$ = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e''$;
(8) l' = sum(l, n') ;
(9) perform $A\ h\ a$ union(f, set of a) $d\ u_n\ u_e$ 0 $l\ l'$ sum(l',2) sum(l',4) =
a-state $p'\ a'_n\ a'_e$;
(10) a'''_n = if z_n then a_n else a'_n ;
(11) a'''_e = if z_e then a_e else a'_e ;
(12) unf $U\ h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z'''_n\ h'''_n\ a'''_n\ z'''_e\ h'''_e\ a'''_e\ e'''$ sum(l',12) $l_n\ l_e\ l_f\ l_u$ =
a-state $p''\ a''_n\ a''_e$;
(13) either(e'' is empty-list, u_n is 0) = true
$\Rightarrow$ unf ⟦ A:Act “or” U:Unf ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$ =
a-state overlay(
p' ,
map of l' to (move a'_n to a''_n),
map of sum(l',1) to (jump l_n),
map of sum(l',2) to (move a'_e to a'_e),
map of sum(l',3) to (jump l_e),
map of sum(l',4) to (load cp -1 commits into global),
map of sum(l',5) to (compare global with 1),
map of sum(l',6) to (branchequal l_f),
combinecommit sum(l',7),
p'')

$a''_n\ a''_e$;

(5) unf ⟦ U:Unf “or” A:Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$ =
unf ⟦ A “or” U ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$;

(6) (1) either(e is empty-list, u_n is 0) = true
$\Rightarrow$ unf “unfold” $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$ = a-state overlay(
map l to (move a to a'), map of sum(l,1) to (jump l_u)) $a_n\ a_e$.

C.4.3 Tuples

needs: **Dependent Data .**

introduces: with _ _ _ _ _ _ _ _ .

- with _ _ _ _ _ _ _ _ :: Tuple, data-type, general-register, frozen, symbol-table, linenumber, linenumber-fail $\longrightarrow$ (program, general-register) (*partial*) .

- h : data-type ;
  r , r' , r'' , a : general-register ;
  f , f' : frozen ;
  d : symbol-table ;
  l , l' , l'' : linenumber ;
  l_f : linenumber-fail ;

$\Rightarrow$

(1) (1) r = free-register f
$\Rightarrow$ with "()" $h\ a\ f\ d\ l\ l_f$ = (empty-list-code $r\ l$, r) ;

(2) (1) d-count $D\ h\ d$ = (n:natural, S:type) ;
(2) l' = sum(l, n) ;
(3) evaluate $D\ h\ a\ f\ d\ l\ l_f$ = (p:program, r:general-register) ;
(4) r' = free-register union(f, set of r)
$\Rightarrow$ with D:Dependent $h\ a\ j\ d\ l\ l_f$ = (overlay(p, single-listcode $r\ r'\ l'$), r') ;

(3) (1) w-count $T\ h\ d$ = wc-state n:natural h':data-type) ;
(2) w-count $T'\ h\ d$ = wc-state n':natural h'':data-type) ;
(3) l' = sum(l, n) ;
(4) l'' = sum(l', n') ;
(5) with $T\ h\ a\ f\ d\ l\ l_f$ = (p:program, r:general-register) ;
(6) f' = union(f, set of(a, r));
(7) with $T'\ h\ a\ f'\ d\ l'\ l_f$ = (p':program, r':general-register) ;
(8) r'' = free-register union($f$set of (r, r'))
$\Rightarrow$ with $[\![T$:Tuple "," T':Tuple $]\!]\ h\ a\ f\ d\ l\ l_f$ =
(overlay(p, p', concatenation-code $r\ r'\ r''\ l''$), r'') ;

(4) with "them" $h\ a\ f\ d\ l\ l_f$ = (empty-map, a) .

C.4.4 Dependent Data

needs: **Actions , Lookup in Symbol Tables ,**

Unary Operations , Binary Operations .

introduces: evaluate _ _ _ _ _ _ _ _ .

- evaluate _ _ _ _ _ _ _ _ ::
  Dependent, data-type, general-register, frozen, symbol-table,
  linenumber, linenumber-fail → (program, general-register) (*partial*) .

h , h' , h_n , h_e : data-type ;
a , a_n , a_e , r , r' : general-register ;
f , f' : frozen ;
d : symbol-table ;
l , l' , l'' : linenumber ;
l_f : linenumber-fail ;
n : natural ;
z_n , z_e : truth-value ;
p : program

⇒

(1) (1) r = free-register f
⇒
(2) evaluate "true" $h\ a\ f\ d\ l\ l_f$ = (map of l to (move 1 to r)), r) ;
(3) evaluate "false" $h\ a\ f\ d\ l\ l_f$ = (map of l to (move 0 to r)), r) ;
(4) evaluate n:natural $h\ a\ f\ d\ l\ l_f$ = (map of l to (move n to r)), r) ;
(5) evaluate ⟦ "empty-list" "&" "[" T:Type "]" "list" ⟧ $h\ a\ f\ d\ l\ l_f$ =
(empty-list-code r l, r) ;

(2) (1) data-type $D = h'$;
(2) a-count A h' concatenation(list of empty-list, d) =
ac-state n z_n h_n z_e h_e empty-list ;
(3) l' = sum(l,1,n) ;
(4) perform A h' (reg 0) empty-set concatenation(list of empty-list, d) 0 0 0
sum(l,1) l' sum(l',2) sum(l',3) = a-state p a_n a_e ;
(5) r = free-register f
⇒ evaluate ⟦ "closure" "abstraction" "of" A:Act "&" "[" "perhaps"

"using" D:Data "]" "act"⟧ $h\ a\ f\ d\ l\ l_f$ = overlay(
map of sum(l,0) to (jump sum(l',4)),
p ,
return-sequence a_n a_e l' ,

```
            map of sum($l'$,4) to ( move sum($l$,1) to global ),
            map of sum($l'$,5) to ( store global in hp 0 heap ),
            map of sum($l'$,6) to ( store staticlink in hp 1 heap ),
            map of sum($l'$,7) to ( move hp to $r$ ),
            map of sum($l'$,8) to ( move sum hp 2 to hp )) ,

            $r$) ;
(3)   (1)   d-count $D\ h\ d$ = ($n$:natural, $S$:type) ;
      (2)   evaluate $D\ h\ a\ f\ d\ l\ l_f$ = ($p$:program, $r$:general-register) ;
      (3)   $r'$ = free-register union($f$, set of $r$)
      ⇒     evaluate ⟦ $O$:Unary $D$:Dependent ⟧ $h\ a\ f\ d\ l\ l_f$ =
                  (overlay($p$, unary-code $O\ r\ r'$ sum($l,n$) $l_f$), $r'$) ;
(4)   (1)   d-count $D\ h\ d$ = ($n'$:natural, $S'$:type) ;
      (2)   d-count $D'\ h\ d$ = ($n''$:natural, $S''$:type) ;
      (3)   $l'$ = sum($l,n'$) ;
      (4)   $l''$ = sum($l',n''$) ;
      (5)   evaluate $D\ h\ a$ union($f$, set of $a$) $d\ l\ l_f$ = ($p'$:program, $r'$:general-register) ;
      (6)   $f'$ = union($f$, set of $r'$) ;
      (7)   evaluate $D'\ h\ a\ f'\ d\ l'\ l_f$ = ($p''$:program, $r''$:general-register) ;
      (8)   $r$ = free-register union($f$, set of ($r'$, $r''$))
      ⇒
      (9)   evaluate ⟦ $O$:Binary "(" $D$:Dependent "," $D'$:Dependent ")" ⟧ $h\ d\ f\ d\ l\ l_f$ =
                  (overlay($p'$, $p''$, binary-code $O\ r'\ r''\ r\ l''\ l_f$), $r$) ;
      (10)  evaluate ⟦ $D$:Dependent $O$:( "is" | ⟦ "is" "less" "than" ⟧ ) $D'$:Dependent⟧
                  $h\ d\ f\ d\ l\ l_f$ =
                  (overlay($p'$, $p''$, binary-code $O\ r'\ r''\ r\ l''\ l_f$), $r$) ;
      (11)  evaluate ⟦ "component#" $D'$:Dependent "items" $D$:Dependent⟧ $h\ d\ f\ d\ l\ l_f$ =
                  (overlay($p'$, $p''$, binary-code "at" $r'\ r''\ r\ l''\ l_f$), $r$) ;
(5)   (1)   $r$ = free-register union($f$, set of $a$) ;
      (2)   $r'$ = free-register union($f$, set of ($a$, $r$) ) ;
      ⇒     evaluate "it" $h\ a\ f\ d\ l\ l_f$ = (overlay(
                  map of $l$ to ( move 1 to $r$ ),
                  at-code $a\ r\ r'$ sum($l$,1) $l_f$),

                  $r'$) ;
(6)   (1)   $r$ = free-register union($f$, set of $a$) ;
      (2)   $r'$ = free-register union($f$, set of )$a$, $r$)) ;
```

```
        ⇒   evaluate ⟦ “the” “given” $S$:Datum “#” $n$:natural ⟧ $h\ a\ f\ d\ l\ l_f$ = (overlay(
                map of $l$ to ( move $n$ to $r$ ),
                at-code $a\ r\ r'$ sum($l$,1) $l_f$),

                $r'$) ;
(7)     (1) find-count $k\ d$ = ($n$:natural, $S'$:type, $j$:natural) ;
        (2) $r$ = free-register $f$ ;
        (3) $l'$ = sum($l$, 1,$n$) ;
        (4) find $k\ d$ sum($l$,1) = $p$:program
        ⇒   evaluate ⟦ “the” $S$:Datum “bound” “to” $k$:token ⟧ $h\ a\ f\ d\ l\ l_f$ = (overlay(
                map of $l$ to ( move statclink to global ),
                $p$,
                map of $l'$ to ( load global $j$ stack into $r$)),

                $r'$) ;
(8)     (1) d-count $D\ h\ d$ = ($n$:natural, $S'$:type) ;
        (2) $l'$ = sum($l, n$) ;
        (3) evaluate $D\ h\ a\ f\ d\ l\ l_f$ = ($p$:program, $r$:general-register) ;
        (4) $r'$ = free-register union($f$,set of $r$)
        ⇒   evaluate ⟦ “the” $S$:Datum “stored” “in” $D$:Dependent ⟧ $h\ a\ f\ d\ l\ l_f$ = (overlay(
                $p$,
                map of sum($l'$, 0) to ( load $r$ 0 store into global ) ,
                map of sum($l'$, 1) to ( compare global with 0 ) ,
                map of sum($l'$, 2) to ( branchequal $l_f$ ) ,
                map of sum($l'$, 3) to ( load $r$ 1 store into $r'$ )) ,

                $r'$) ;
(9)     evaluate ⟦ “(” $D$:Dependent “)” ⟧ $h\ a\ f\ d\ l\ l_f$ = evaluate $D\ h\ a\ f\ d\ l\ l_f$ .
```

C.4.5 Lookup in Symbol Tables

introduces: find _ _ _ .

- find _ _ _ :: token, symbol-table, linenumber → (program) (*partial*) .

(1) find k:token concatenation(list of e:block, d:symbol-table) l:linenumber =

```
            if (block-find-count $k$ $e$) is error
            then overlay( map $l$ to ( load global 0 stack into global ),
                  find $k$ $d$ sum($l$,1))
            else empty-map .
```

C.4.6 Unary Operations

introduces: unary-code _ _ _ _ _ .

```
•     unary-code _ _ _ _ _ :: Unary, general-register, general-register, linenumber
            linenumber-fail → (program) ($partial$) .
      $r$, $r'$ : general-register ;
      $l$ : linenumber ;
      $l_f$ : linenumber-fail
⇒
(1)   unary-code "not" $r$ $r'$ $l$ $l_f$ = overlay(
                  map of sum($l$,0) to ( move 1 to global )
                  map of sum($l$,1) to ( move difference global $r$ to $r'$ ),
(2)   unary-code "negation" $r$ $r'$ $l$ $l_f$ = overlay(
                  map of sum($l$,0) to ( move 0 to global )
                  map of sum($l$,1) to ( move difference global $r$ to $r'$ ) ;
(3)   unary-code ⟦ "list" "of" ⟧ $r$ $r'$ $l$ $l_f$ = single-list-code $r$ $r'$ $l$ ;
(4)   unary-code "head" $r$ $r'$ $l$ $l_f$ = head-code $r$ $r'$ $l$ $l_f$;
(5)   unary-code "tail" $r$ $r'$ $l$ $l_f$ = tail-code $r$ $r'$ $l$ $l_f$;
```

C.4.7 Binary Operations

introduces: binary-code _ _ _ _ _ _ .

```
•     binary-code _ _ _ _ _ _:: Binary | "is" | ⟦ "is" "less" "than" ⟧ | "at",
            general-register, general-register, general-register,
            linenumber, linenumber-fail → (program) ($partial$) .

      $r$, $r'$ $r''$ : general-register ;
      $l$ : linenumber ;
```

```
        $l_f$ : linenumber-fail
$\Rightarrow$
(1)     binary-code "both" $r'$ $r''$ $r$ $l$ $l_f$ = overlay(
                    map of sum($l$,0) to ( compare $r'$ with 0 ),
                    map of sum($l$,1) to ( branchequal sum($l$,6) ),
                    map of sum($l$,2) to ( compare $r''$ with 0 ),
                    map of sum($l$,3) to ( branchequal sum($l$,6) ),
                    map of sum($l$,4) to ( move 1 to $r$ ),
                    map of sum($l$,5) to ( jump sum($l$,7) ),
                    map of sum($l$,6) to ( move 0 to $r$ )) ;
(2)     binary-code "either" $r'$ $r''$ $r$ $l$ $l_f$ = overlay(
                    map of sum($l$,0) to ( compare $r'$ with 1 ),
                    map of sum($l$,1) to ( branchequal sum($l$,6) ),
                    map of sum($l$,2) to ( compare $r''$ with 1 ),
                    map of sum($l$,3) to ( branchequal sum($l$,6) ),
                    map of sum($l$,4) to ( move 0 to $r$ ),
                    map of sum($l$,5) to ( jump sum($l$,7) ),
                    map of sum($l$,6) to ( move 1 to $r$ )) ;
(3)     binary-code "sum" $r'$ $r''$ $r$ $l$ $l_f$ =
                    map of $l$ to ( move sum $r'$ $r''$ to $r$ ) ;
(4)     binary-code "difference" $r'$ $r''$ $r$ $l$ $l_f$ =
                    map of $l$ to ( move difference $r'$ $r''$ to $r$ ) ;
(5)     binary-code "concatenation" $r'$ $r''$ $r$ $l$ $l_f$ = concatenation-code $r'$ $r''$ $r$ $l$ ;
(6)     binary-code "is" $r'$ $r''$ $r$ $l$ $l_f$ = overlay(
                    map of sum($l$,0) to ( compare $r'$ with $r''$ ),
                    map of sum($l$,1) to ( branchequal sum($l$,4) ),
                    map of sum($l$,2) to ( move 0 to $r$ ),
                    map of sum($l$,3) to ( jump sum($l$,5) ),
                    map of sum($l$,4) to ( move 1 to $r$ ),
(7)     binary-code $[\![$ "is" "less" "than" $]\!]$ $r'$ $r''$ $r$ $l$ $l_f$ = overlay(
                    map of sum($l$,0) to ( compare $r'$ with $r''$ ),
                    map of sum($l$,1) to ( branchequal sum($l$,4) ),
                    map of sum($l$,2) to ( move 0 to $r$ ),
                    map of sum($l$,3) to ( jump sum($l$,5) ),
                    map of sum($l$,4) to ( move 1 to $r$ ),
(8)     binary-code "at" $r'$ $r''$ $r$ $l$ $l_f$ = at-code $r'$ $r''$ $r$ $l$ $l_f$ .
```

Appendix D

Abstraction of Semantic Entities

needs: **Data Notation ,**
A Pseudo SPARC Machine Language/Abstract Syntax .

D.1 Auxiliary Notation

introduces: _ is submap of _ , _ is _ .

gramma:

- _ is submap of _ :: program, program $\rightarrow$ truth-value (*total*) .
- _ is _ :: instruction, instruction $\rightarrow$ truth-value (*total, commutative*) .
- _ is _ :: movable, movable $\rightarrow$ truth-value (*total, communative*) .
- _ is _ :: argument, argument $\rightarrow$ truth-value (*total, communative*) .
- _ is _ :: register, register $\rightarrow$ truth-value (*total, comutative*) .
- _ is _ :: page-id, page-id $\rightarrow$ truth-value (*total, commutative*) .

p , p' : program ;
i , i' : integer ;
R , R' , R'' , R''' : register ;
P : page-id ;
M , M' : movable ;
A , A' : argument ;

n , n' : natural

$\Rightarrow$

(1) p is submap of p' = overlay(p, p') is p' ;

(2) skip is I:(call | return | storeregisters | loadregisters | (jump i) |
(branchequal i) | (branchlessthan i) | (store R in R' i P) |
(load R i P into R') | (move M to R) | (compare R with A)) = false ;

(3) call is I:(return | storeregisters | loadregisters | (jump i) |
(branchequal i) | (branchlessthan i) | (store R in R' i P) |
(load R i P into R') | (move M to R) | (compare R with A)) = false ;

(4) return is I:(store registers | loadregisters | (jump i) |
(branchequal i) | (branchlessthan i) | (store R in R' i P) |
(load R i P into R') | (move M to R) | (compare R with A)) = false ;

(5) storeregisters is I:(loadregisters | (jump i) | (branchequal i) | (branchlessthan i) |
(store R in R' i P) | (load R i P into R') | (move M to R) |
(compare R with A)) = false ;

(6) loadregisters is I:((jump i) | (branchequal i) | (branchlessthan i) |
(store R in R' i P) | (load R i P into R') | (move M to R) |
(compare R with A)) = false ;

(7) (jump i') is I:((branchequal i) | (branchlessthan i) | (store R in R' i P) |
(load R i P into R') | (move M to R) | (compare R with A)) = false ;

(8) (branchequal i') is I:((branchlessthan i) | (store R in R' i P) |
(load R i P into R') | (move M to R) | (compare R with A)) = false ;

(9) (branchlessthan i') is I:((store R in R' i P) | (load R i P into R') |
(move M to R) | (compare R with A)) = false ;

(10) (store R in R' i P) is I:((load R'' i' P' into R''') | (move M' to R'') |
(compare R'' with A')) = false ;

(11) (load R i P into R') is I:((move M' to R'') | (compare R'' with A')) $=$ false ;

(12) (move M to R) is (compare R' with A) $=$ false ;

(13) (jump i) is (jump i') $= i$ is i' ;

(14) (branchequal i) is (branchequal i') $= i$ is i' ;

(15) (branchlessthan i) is (branchlessthan i') $= i$ is i' ;

(16) (store R in R' i P) is (store R'' in R''' i' P') $=$
all(R is R'', R' is R''', i is i', P is P') ;

(17) (load R i P into R') is (load R'' i' P' into R''') $=$
all(R is R'', R' is R''', i is i', P is P') ;

(18) (move M to R) is (move M' to R') $=$ both(M is M', R is R') ;

(19) (compare R with A) is (compare R' with A') $=$ both(R is R', A is A') ;

(20) (sum R A) is (difference R' A') $=$ false ;

(21) (sum R A) is (sum R' A') $=$ both(R is R', A is A') ;

(22) (difference R A) is (difference R' A') $=$ both(R is R', A is A') ;

(23) R:register is i:integer $=$ false ;

(24) firstfree is x:(sp | hp | cp | cef | global | arg | staticlink | (reg n)) $=$ false ;

(25) sp is x:(hp | cp | cef | global | arg | staticlink | (reg n)) $=$ false ;

(26) hp is x:(cp | cef | global | arg | staticlink | (reg n)) $=$ false ;

(27) cp is x:(cef | global | arg | staticlink | (reg n)) $=$ false ;

(28) cef is x:(global | arg | staticlink | (reg n)) $=$ false ;

(29) global is x:(arg | staticlink | (reg n)) = false ;

(30) arg is x:(staticlink | (reg n)) = false ;

(31) staticlink is (reg n) = false ;

(32) (reg n) is (reg n') = n is n' ;

(33) stack is x:(store | heap | commits) = false ;

(34) store is x:(heap | commits) = false ;

(35) heap is commits = false .

D.2 Abstraction Functions

needs: **Data Notation ,**
A Compilable Subset of Action Notation/Abstract Syntax ,
A Compilable Subset of Action Notation/Semantic Entities ,
A Pseudo SPARC Machine Language/Abstract Syntax ,
A Pseudo SPARC Machine Language/Semantic Entities ,
Actions to SPARC Compiler ,
Auxiliary Notation .

introduces: t-abs _ _ _ _ , b-abs _ _ _ _ , e-abs _ _ _ _ , store-abs _ _ ,
s-abs _ _ , i-abs _ _ , o-abs _ _ , c-abs _ ,
m-abs _ _ _ _ _ _ _ _ _ _ _ , v-abs _ _ _ _ , storable-abs _ .

- t-abs _ _ _ _ _ :: natural, memory, program, data-type $\rightarrow$ data .
- b-abs _ _ _ _ :: natural, memory, program, symbol-table $\rightarrow$ bindings .
- e-abs _ _ _ _ :: natural, memory, program, block $\rightarrow$ bindings .
- store-abs _ _ :: natural, page $\rightarrow$ storage .

- s-abs _ _ :: natural, page $\longrightarrow$ storage-map .

- i-abs _ _ :: naturals page $\longrightarrow$ [integer] list .

- o-abs _ _ :: natural, page $\longrightarrow$ [integer] list .

- c-abs _ :: integer $\longrightarrow$ truth-value ($partial$) .

- m-abs _ _ _ _ _ _ _ _ _ _ _ ::
  spare-state, truth-value, data-type, truth-value, data-type,
  general-register, general-register, block,
  linenumber-complete, linenumber-escape, linenumber-fail $\longrightarrow$ state .

- v-abs _ _ _ _ :: integer, memory, program, type $\longrightarrow$ datum .

- storable-abs _ :: naturals page $\longrightarrow$ storage-map .

n : natural ;
q : memory ;
p , p' , p'' : program ;
S : type ;
h , h_n , h_e : data-type ;
e : block ;
b , b' : bindings ;
v , v' : datum ;
m , m' : storage-map ;
k : token :
se : page ;
cz : was-zero ;
cn : was-negative ;
z_n , z_e : truth-value ;
d : symbol-table ;
s : storage :
g : globals ;
w : windows ;
a_n , a_e : general-register ;
l , l' : linenumber ;
l_n : linenumber-complete ;

l_e : linenumber-escape ;
l_f : linenumber-fail ;
c : commitment

$\Rightarrow$

(1) (1) (q at heap) at sum(n,1) = -1
$\Rightarrow$ t-abs n q p () :- () ;

(2) (1) v-abs ((q at heap) at n) q p S :- v ;
(2) t-abs ((q at heap) at sum(n, 1)) q p h :- t ;
$\Rightarrow$ t-abs n q p (S, h) :- (v,t) ;

(3) b-abs n q p empty-list :- empty-map;

(4) (1) e-abs sum(n,count of items of e) q p e :- b ;
(2) b-abs ((q at stack) at n) q p d :- b'
$\Rightarrow$ b-abs n q p concatenation(list of e, d) :- overlay(b, b') ;

(5) e-abs n q p empty-list :- empty-map ;

(6) (1) v-abs ((q at stack) at successor(n)) q p S :- v ;
(2) e-abs n q p e :- b
$\Rightarrow$ e-abs successor(n) q p concatenation(list of (entry(k, S), e) :-
overlay(map k to v, b) ;

(7) (1) s-abs n se :- m
$\Rightarrow$ store-abs n:natural se:page :- (m,n) ;

(8) s-abs 0 empty-map :- empty-map ;

(9) (1) storable-abs n se :- m ;
(2) s-abs n se :- m' ;
$\Rightarrow$ s-abs sum(n,2) se :- overlay(m, m') ;

(10) i-abs 0 se = empty-list ;

(11) (1) se at successor(n) = i:integer ;
(2) i-abs n se = il:[integer] list
$\Rightarrow$ i-abs successor(n) se = concatenation(list of i, il) ;

(12) o-abs 0 se = empty-list ;

(13) (1) se at successor(n) = i:integer ;
(2) o-abs n se = i:[integer] list
$\Rightarrow$ o-abs successor(n) se = concatenation(ol, list of i) ;

(14) c-abs i:integer = if i is 0 then false else

```
            if $i$ is 1 then true else nothing ;
(15)  (1)   store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
      (2)   $io$:input-output = ($il$:[integer] list, $ol$:[integer] list) ;
      (3)   i-abs (($q$ at input) at 0) ($q$ at input) = $il$ ;
      (4)   o-abs (($q$ at output) at 0) ($q$ at output) = $ol$
      $\Rightarrow$
      (5)   (1) ($g$ at cef) is 0 = true ;
            (2) t-abs ((head of $w$) at $a_n$) $q$ $p$ $h_n$ :- $t$ ;
            (3) e-abs difference($g$ at sp, count of items of $e$) $q$ $p$ $e$ :- $b$
            $\Rightarrow$m-abs ($p$, $l_n$, $cz$, $cn$, $g$, $w$, $q$) true $h_n$ $z_e$ $h_e$ $a_n$ $a_e$ $e$ $l_n$ $l_e$ $l_f$ :-
                  completed $t$ $b$ $s$ $io$ $c$ ;
      (6)   (1) ($g$ at cef) is 1 = true ;
            (2) t-abs ((head of $w$) at $a_e$) $q$ $p$ $h_e$ :- $t$
            $\Rightarrow$m-abs ($p$, $l_e$, $cz$, $cn$, $g$, $w$, $q$) $z_n$ $h_n$ true $h_e$ $a_n$ $a_e$ $e$ $l_n$ $l_e$ $l_f$ :-
                  escaped $t$ $s$ $io$ $c$ ;
      (7)   (1) ($g$ at cef) is 2 = true
            $\Rightarrow$m-abs ($p$, $l_f$, $cz$, $cn$, $g$, $w$, $q$) $z_n$ $h_n$ $z_e$ $h_e$ $a_n$ $a_e$ $e$ $l_n$ $l_e$ $l_f$ :-
                  failed $s$ $io$ $c$ ;
(16)  v-abs 0 $q$ $p$ truth-value-type :- false ;
(17)  v-abs 1 $q$ $p$ truth-value-type :- true ;
(18)  v-abs $i$:integer $q$ $p$ integer-type :- $i$ ;
(19)  v-abs $n$:natural $q$ $p$ truth-value-cell-type :- truth-value-cell $n$ ;
(20)  v-abs $n$:natural $q$ $p$ integer-cell-type :- integer-cell $n$ ;
(21)  (1)   a-count $A$ $h$ concatenation(list of empty-list, $d$) =
               ac-state $n$ $z_n$ $h_n$ $z_e$ $h_e$ empty-list ;
      (2)   $l$ = ($q$ at heap) at $i$ ;
      (3)   $l'$ = sum($l$, $n$) ;
      (4)   perform $A$ $h$ (reg 0) empty-set concatenation(list of empty-list, $d$) 0 0 0
               $l$ $l'$ sum($l'$,2) sum($l'$,3) = a-state $p'$ $a_n$ $a_e$ ;
      (5)   return-sequence $a_n$ $a_e$ $l'$ = $p''$ ;
      (6)   $p'$ is submap of $p$ = true ;
      (7)   $p''$ is submap of $p$ = true ;
      (8)   data-type $D$ = $h$ ;
```

(9) b-abs ((*q* at heap) at sum(*i*,1)) *q* *p* *d* :- *b*

⇒ v-abs *i*:integer *q* *p* (abstraction-type *h* z_n h_n z_e h_e *d*) :-
  closure-abstraction *A*:Act *D*:Data *b*:bindings ;

(22) (1) ((*q* at heap) at sum(*i*,1)) is -1 = true

⇒ v-abs *i*:integer *q* *p* (list-type *S*) :- empty-list ;

(23) (1) v-abs ((*q* at heap) at *i*) *q* *p* *S* :- *v* ;

(2) v-abs ((*q* at heap) at sum(*i*,1)) *q* *p* (list-type *S*:type) :- v'

⇒ v-abs *i*:integer *q* *p* (list-type *S*) :- concatenation(list of *v*, v') ;

(24) (1) *se* at *n* = 0

⇒ storable-abs *n* *se* :-
  (map of *x*:((truth-value-cell *n*) | (integer-cell *n*)) to uninitialized) ;

(25) (1) *se* at *n* = 1 ;

(2) *se* at sum(*n*,1) = 0

⇒ storable-abs *n* *se* :- (map of truth-value-cell *n* to false) ;

(26) (1) *se* at *n* = 1 ;

(2) *se* at sum(*n*,1) = 1

⇒ storable-abs *n* *se* :- (map of truth-value-cell *n* to true) ;

(27) (1) *se* at *n* = 1

⇒ storable-abs *n* *se* :- (map of integer-cell *n* to (*se* at sum(*n*,1))) .

Appendix E

Lemmas

needs: **Data Notation ,**
A Compilable Subset of Action Notation ,
A Pseudo SPARC Machine Language ,
Actions to SPARC compiler ,
Abstraction of Semantic Entities .

E.1 Auxiliary Notation

E.1.1 Code Placement

introduces: well-placed _ _ _ ,
a-consistent _ _ _ _ , d-consistent _ _ _ _ , w-consistent _ _ _ _ ,
find-consistent _ _ _ , operation-consistent _ _ _ ,

- well-placed _ _ _ :: program, linenumber, natural → truth-value (*total*) .
- a-consistent _ _ _ _ ::
  ac-state, a-state, cleanup, linenumber → truth-value (*total*) .
- d-consistent _ _ _ _ :: (natural, type) | error, (program,general-register), frozen, linenumber → truth-value (*total*) .
- w-consistent _ _ _ _ :: wc-state | error, (program, general-register), frozen, linenumber → truth-value (*total*) .
- find-consistent _ _ _ :: (naturaltypenatural) | error, program, linenumber → truth-value (*total*) .

- operation-consistent _ _ _ ::
  (natural,type) | error, program, linenumber $\rightarrow$ truth-value ($total$) .

  p : program ;
  l : linenumber ;
  n : natural ;
  z_n , z_e : truth-value ;
  h_n , h_e : data-type ;
  e : block ;
  a_n , a_e : general-register ;
  u_n : cleanup ;
  f : frozen ;
  y : (program, general-register)

$\Rightarrow$

(1) well-placed p l n = if p is empty-map then n is 0 else all(
(count of elements of mapped-set of p) is n,
(minimum of mapped-set of p) is n,
(maximum of mapped-set of p) is difference(sum(l,n),1) ;

(2) a-consistent (ac-staten n z_n h_n z_e h_e e) (a-state p a_n a_e) u_n l = both(
well-placed p l n,
either(e is empty-list, u_n is 0)) ;

(3) d-consistent (x : (natural, type) | error) y f l =
if x is error then true else both(
well-placed (component#1 of y) l (component#1 of x),
(component#2 of y) is not in f) ;

(4) w-consistent (x : wc-state | error) y f l =
if x is error then true else both(
well-placed (component#1 of y) l (code-size of x),
(component#2 of y) is not in f) ;

(5) find-consistent (x : (natural, type, natural) | error) p l =
if x is error then true else
(well-placed p l (component #1 of x)) ;

(6) operation-consistent (x : (natural, type) | error) p l =
if x is error then true else (well-placed p l (component#1 of x)) .

E.1.2 Semantics of Types

introduces: tc-abs _ , bc-abs _ , ec-abs _ , mc-abs _ , vc-abs _ ,
ac-less _ _ _ _ _ _ _ _ _ _ , cc-less _ _ _ _ _ , ec-less _ _ _ _ _ ,

- tc-abs _ :: data-type → data .
- bc-abs _ :: symbol-table → bindings .
- ec-abs _ :: block → bindings .
- mc-abs _ :: ac-state → state .
- vc-abs _ :: type → datum .
- ac-less _ _ _ _ _ _ _ _ _ _ ::
  truth-value, data-type, truth-value, data-type, block,
  truth-value, data-type, truth-value, data-type, block →
  truth-value (*total*) .

  cc-less _ _ _ _ _ ::
  truth-value, data-type, block, truth-value, data-type, block → truth-value (*total*) .

  ec-less _ _ _ _ _ :: truth-value, data-type, truth-value, data-type → truth-value (*total*) .

t, t' : data ;
b, b' : bindings ;
s : storage ;
io : input-output ;
c : commitment ;
h, h', h_n, h'_n, h_e, h'_e : data-type ;
e, e' : block ;
z_n, z'_n, z_e, z'_e : truth-value ;

⇒

(1) tc-abs () = () ;

(2) (1) vc-abs S :- v
⇒ tc-abs S:type :- v ;

(3) (1) tc-abs h :- t ;
(2) tc-abs h' :- t'
⇒ tc-abs (h:data-type, h':data-type) :- (t,t') ;

(4) bc-abs empty-list = empty-map ;

(5) (1) ec-abs e :- b
⇒ bc-abs list of e:block :- b ;

(6) (1) bc-abs d :- b ;
(2) bc-abs d' :- b'
⇒ bc-abs concatenation(d:symbol-table, d':symbol-table) :- overlay(b, b') ;

(7) ec-abs empty-list = empty-map ;

(8) (1) vc-abs S :- v
⇒ ec-abs list of entry k:token S:type :- map of k to v ;

(9) (1) ec-abs e :- b ;
(2) ec-abs e' :- b'
⇒ ec-abs concatenation(e:block, e':block) :- overlay(b, b') ;

(10) mc-abs x:ac-state :- failed s io c ;

(11) (1) tc-abs h_e :- t
⇒ mc-abs (ac-state n:natural z_n:truth-value h_n:data-type true h_e:data-type e:block) :- escaped t s io c ;

(12) (1) tc-abs h_n :- t ;
(2) ec-abs e' :- b
⇒ mc-abs (ac-state n:natural true h_n:data-type z_e:truth-value h_e:data-type e:block) :-completed t b s io c ;

(13) vc-abs truth-value-type = truth-value ;

(14) vc-abs integer-type = integer ;

(15) vc-abs truth-value-cell-type = truth-value-cell ;

(16) vc-abs integer-cell-type = integer-cell ;

(17) (1) a-count A h concatenation(list of empty-list, d) =
ac-state n:natural z_n h_n z_e h_e empty-list ;
(2) data-type $D = h$;
(3) bc-abs d :- b
⇒ vc-abs (abstraction-type h:data-type z_n:truth-value h_n:data-type z_e:truth-value $h_e$data-type $d$symbol-table) :-
closure-abstraction A Act D:Data b:bindings ;

(18) vc-abs (list-type S:type) = [vc-abs S] list ;

(19) ac-less z_n h_n z_e h_e e z'_n h'_n z'_e h'_e e'
= both(cc-less z_n h_n e z'_n h'_n e', ec-less z_e h_e z'_e h'_e) ;

(20) cc-less false h e false h' e' = true ;

(21) cc-less true h e false h' e' = false ;

(22) cc-less false h e true h' e' = true ;

(23) cc-less true h e true h' e' = both(h is h', e is e') ;

(24) ec-less false h false h' = true ;

(25) ec-less true h false h' = false ;

(26) ec-less false h true h' = true ;

(27) ec-less true h true h' = h is h' ;

E.1.3 Program Execution

introduces: _ leq _ , cleaned-up _ _ _ _ _ _ , up-to _ ,
q-earlier _ _ _ _ , m-earlier _ _ , mq-earlier _ _ ,
basic-pre-condition _ _ _ _ _ _ _ _ _ , pre-condition _ _ _ _ _ _ _ _ _ _ ,
basic-post-condition _ _ _ _ _ _ _ _ _ ,
a-post-condition _ _ _ _ _ _ _ _ _ _ ,
u-post-condition _ _ _ _ _ _ _ _ _ _ ,
unchanged _ _ _ _ _ ,
post-condition _ _ _ _ ,
d-post-condition _ _ _ _ _ , f-post-condition _ _ _ _ ,
op-post-condition _ _ _ _ _ , .

- _ leq _ :: natural, natural* → truth-value(*total*) .
- cleaned-up _ _ _ _ _ _ ::
  integer, integer, natural, clean-up, clean-up, clean-up → integer
  (*partial*) .
- up-to _ :: natural → [natural] set .
- q-earlier _ _ _ _ ::
  natural, natural, memory, memory → truth-value (*total*) .
- m-earlier _ _ :: sparc-state, sparc-state → truth-value (*total*) .
- mq-earlier _ _ ::
  sparc-state, memory → truth-value (*total*) .
- basic-precondition _ _ _ _ _ _ _ _ _ :: program, program,
  linenumber, linenumber*, globals, windows, memory,
  memory, natural → truth-value (*total*) .
- precondition _ _ _ _ _ _ _ _ _ _ :: program, program,

linenumber, linenumber*, globals, windows, memory,
memory, natural, symbol-table → truth-value $(total)$.

- basic-post-condition _ _ _ _ _ _ _ _ _ _ :: globals, windows, memory, sparc-state, frozen, cleanup, cleanup, cleanup, block → truth-value $(total)$.
- a-post-condition _ _ _ _ _ _ _ _ _ _ :: globals, windows, memory, sparc-state, frozen, cleanup, cleanup, cleanup, block, commitment → truth-value $(partial)$.
- u-post-condition _ _ _ _ _ _ _ _ _ _ :: globals, windows, memory, sparc-state, frozen, cleanup, cleanup, cleanup, block, commitment → truth-value $(total)$.
- unchanged _ _ _ _ _ ::
  natural, windows, memory, windows, memory → truth-value $(total)$.
- post-condition _ _ _ _ ::
  globals, windows, memory, sparc-state → truth-value $(total)$.
- d-post-condition _ _ _ _ _ ::
  globals, windows, memory, sparc-state, frozen
  → truth-value $(partial)$.
- f-post-condition _ _ _ _ ::
  globals, windows, memory, sparc-state → truth-value $(total)$.
- op-post-condition _ _ _ _ _ ::
  globals, windows, memory, sparc-state, frozen
  → truth-value $(total)$.

n, n', nw : natural ;
nt, nt' : natural* ;
sp, cef : integer ;
u_n, u_e, u_f : cleanup ;
e : block ;
c : commitment ;
w, w' : windows ;
q, q' : memory ;
m_p, m'_p : sparc-state ;
p, p' : program ;
g : globals ;
l : linenumber ;
lt : linenumber* ;

```
    $d$ : symbol-table ;
    $f$ : frosen ;
    $r$ : general-register ;

$\Rightarrow$

(1) $n$ leq () = true ;
(2) $n$ leq $n'$:natural = $n$ is less than successor($n'$) ;
(3) $n$ leq $(nt, nt')$ = both($n$ leq $nt$, $n$ leq $nt'$) ;
(4) cleaned-up $sp$ $cef$ $n$ $u_n$ $u_e$ $u_f$ =
        if $cef$ is 0 then difference(sum($sp$,$n$),$u_n$ else
        if $cef$ is 1 then difference(sum($sp$,$u_e$) else
        if $cef$ is 2 then difference(sum($sp$,$u_f$) else nothing ;
(5) up-to 0 = empty-set ;
(6) up-to successor $n$ = union(up-to $n$, $n$) ;
(7) q-earlier $n$ $n'$ $q$ $q'$ = all(
        (($q$ at stack) restricted to up-to successor $n$) is
            (($q'$ at stack) restricted to up-to successor $n$),
        ($q$ at storage) is ($q'$ at storage),
        (($q$ at heap) restricted to up-to $n'$) is
            (($q'$ at heap) restricted to up-to $n'$),
        ($q$ at commits) is ($q'$ at commits),
        ($q$ at input) is ($q'$ at input),
        ($q$ at output) is ($q'$ at output)) ;
(8) m-earlier $m_p$ $m'_p$ = all(
        (program of $m_p$) is (program of $m'_p$),
        (program-counter of $m_p$) is (program-counter of $p'_m$),
        (was-zero of $m_p$) is (was-zero of $m'_p$),
        (was-negative of $m_p$) is (was-negative of $m'_p$),
        (globals of $m_p$) is (globals of $m'_p$),
        (windows of $m_p$) is (windows of $m'_p$),
        q-earlier ((globals of $m_p$) at sp) ((globals of $m'_p$) at hp)
            (memory of $m_p$) (memory of $m'_p$) ;
(9) m-earlier $m_p$ $q$ =
        q-earlier ((globals of $m_p$ at sp) ((globals of $m_p$) at hp)
            (memory of $m_p$) $q$ ;
```

```
(10) basic-precondition $p'$ $p$ $l$ $lt$ $g$ $w$ $q$ $q'$ $nw$ = all(
         $p'$ is submap of $p$,
         $l$ leq $lt$,
         (count of items of $w$) is $nw$,
         q-earlier ($g$ at sp) ($g$ at hp) $q$ $q'$) ;
(11) pre-condition $p'$ $p$ $l$ $lt$ $g$ $w$ $q$ $q'$ $nw$ $d$ = all(
         basic-pre-condition $p'$ $p$ $l$ $lt$ $g$ $w$ $q$ $q'$ $nw$,
         ($g$ at sp) is sum((head of $w$) at staticlink, count of items of head of $d$)) ;
(12) basic-post-condition $g$ $w$ $q$ $m_p$ $f$ $u_n$ $u_e$ $u_f$ $e$ = all(
         ((globals of $m_p$) at sp) is (cleaned-up ($g$ at sp) ((globals of $m_p$) at cef)
             (count of items of $e$) $u_n$ $u_e$ $u_f$),
         ((head of $w$) restricted to $f$) is
             ((head of windows of $m_p$) restricted to $f$),
         q-earlier ($g$ at sp) ($g$ at hp) $q$ (memory of $m_p$),
         ((globals of $m_p$) at cp) is successor($g$ at cp),
         (tail of $w$) is (tail of windows of $m_p$),
         ((head of $w$) restricted to set of (return-address, staticlink) is
             (head of windows of $m_p$) restricted to set of (return-address,
                 staticlink))) ;
(13) a-post-condition $g$ $w$ $q$ $m_p$ $f$ $u_n$ $u_e$ $u_f$ $e$ $c$ = all(
         basic-post-condition $g$ $w$ $q$ $m_p$ $f$ $u_n$ $u_e$ $u_f$ $e$,
         (($q$ at commits) restricted to up-to($g$ at cp)) is
             ((($m_p$ at memory) at commits) restricted to up-to ($g$ at cp)),
         (cabs ((memory of $m_p$) at commits) at
             (predecessor of ((globals of $m_p$) at cp))) is $c$) ;
(14) u-post-condition $g$ $w$ $q$ $m_p$ $f$ $u_n$ $u_e$ $u_f$ $e$ $c$ = all(
         basic-post-condition $g$ $w$ $q$ $m_p$ $f$ $u_n$ $u_e$ $u_f$ $e$,
         (($q$ at commits) restricted to up-to the predecessor of ($g$ at cp)) is
             (((memory of $m_p$) at commits) restricted to
             up-to the predecessor of ($g$ at cp)),
         either(c-abs ((memory of $m_p$) at commits) at
             (predecessor of ((globals of $m_p$) at cp)),
             c-abs ((memory of $m_p$) at commits) at
             (predecessor of predecessor of ((globals of $m_p$) at cp))) is
             either($c$, c-abs ($q$ at commits) at (predecessor of ($g$ at cp)))) ;
(15) unchanged $n$ $w$ $q$ $w'$ $q'$ = all(
         (tail of $w$) is (tail of $w'$),
```

((head of w) restricted to union(set of return-address, set of staticlink)) is
((head of w') restricted to union(set of return-address, set of staticlink)),
(q at stack) is (q' at stack),
(q at storage) is (q' at storage),
((q at heap) restricted to up-to n) is ((q' at heap) restricted to up-to n),
(q at commits) is (q' at commits),
(q at input) is (q' at input),
(q at output) is (q' at output)) ;

(16) post-condition $g\ w\ q\ m_p$ = all(
unchanged (g at hp) $w\ q$ (windows of m_p) (memory of m_p),
(g at sp) is ((globals of m_p) at sp),
(g at firstfree) is ((globals of m_p) at firstfree),
(g at op) is ((globals of m_p at cp)) ;

(17) d-post-condition $g\ w\ q\ m_p\ f$ = both(
((head of w) restricted to f) is ((head of windows of m_p) restricted to f),
post-condition $g\ w\ q\ m_p$) ;

(18) f-post-condition $g\ w\ q\ m_p$ = both(
(head of w) is (head of windows of m_p)
post-condition $g\ w\ q\ m_p$) ;

(19) op-post-condition $g\ w\ q\ m_p\ r$ = both(
((head of w) omitting (set of r)) is ((head of windows of m_p) omitting (set of r)),
post-condition $g\ w\ q\ m_p$) .

E.2 Compiler Consistency

needs: Auxiliary Notation .

Lemma: (Calculation of Free Registers)

(1) (free-register f) is not in f:frozen = true .

Proof: Consider the definition of 'free-register f' in C.l.3.(15) that is

- free-register f:frozen = if f is empty-set then reg 0 else
  minimum of difference(registers up to successor(maximum of f), f) .

There are two cases. If f is the empty set, then 'free-register f' is not contained in it. If f is non-empty, then 'maximum of f' is an individual. In this case, 'difference(registers up to successor(maximum of f), f)' is an individual disjoint from f, so its minimum is not contained in f. □

Lemma: (Code Macro Size)

r, r', r'' : general-register ;
l, l' : linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
u : cleanup

$\Rightarrow$

(1) well-placed (empty-list-code r l) l e-size = true ;

(2) well-placed (single-list-code r r' l) l s-size = true ;

(3) well-placed (concatenation-code r r' r'' l) l c-size = true ;

(4) well-placed (head-code r r' l l_f) l h-size = true ;

(5) well-placed (tail-code r r' l l_f) l t-size = true ;

(6) well-placed (at-code r r' r'' l l_f) l a-size = true ;

(7) well-placed (putcommit l (i/textsf:0 | 1) l 3 = true ;

(8) well-placed (combinecommit l) l 5 = true ;

(9) well-placed (combine l l_n l_e l_f) l 18 = true ;

(10) well-placed (finalize l u(i:0 | 1 | 2) l') l 3 = true ;

(11) well-placed (call-sequence l r r' u_n u_e u_f l_n l_e l_f) l 27 = true ;

(12) well-placed (return-sequence a_n a_e l) l 4 = true ;

Proof: We will prove the first point in detail, the others can be proved similarly. Consider the definition of 'empty-list-code r l' in C.2.1.(1) that is

- empty-list-code r l = overlay(

map of sum(l,0) to (move -1 to global) ,
map of sum(l,1) to (store global in hp 1 heap) ,
map of sum(l,2) to (move hp to r) ,
map of sum(l,3) to (move sum hp 2 to hp)) .

Consider also the definition of 'well-placed p l n' in E.1.1.(1) that is

- well-placed p l n= if p is empty-map then n is 0 else all(
  (count of elements of mapped-set of p) is n ,
  (minimum of mapped-set of p) is l,
  (maximum of mapped-set of p) is difference(sum(l,n),1)) .

The program p is not the empty map, so there are three things to prove. Firstly, there are four elements in the domain of p, and in the definition of 'e-size' in C.2.1.(7) we find that 'e-size = 4'. Secondly, the minimal element in the domain of p is l. Thirdly, the maximal element in the domain of p is 'sum(l,3)', which equals 'difference(sum(l,4),1))'. □

Lemma: (Compiler Consistency)

a , a' , a_n , a_e , r , r' , r'' : general-register ;
u_n , u_e , u_f : cleanup ;
l : linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
l_u : linenumber-unfold ;
h , h_n , h_e : data-type ;
d : symbol-table ;
f : frosen ;
z_n , z_e : truth-value ;
e : block ;
S , S' : type ;

$\Rightarrow$

(1) (1) a-count A:Act h d = x:ac-state ;
(2) perform A h a f d u_n u_e u_f l l_n l_e l_f = y:a-state

$\Rightarrow$ a-consistent $x\ y\ u_n\ l$ = true ;

(2) (1) u-count U:Unf $h\ d\ z_n\ h_n\ z_e\ h_e\ e = x$:ac-state ;

(2) unf $U\ h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u = y$:a-state

$\Rightarrow$ a-consistent $x\ y\ u_n\ l$ = true ;

(3) (1) d-count D:Dependent $h\ d = x$:(natural, type) | error ;

(2) evaluate $U\ h\ a\ f\ d\ l\ l_f = y$:(program, general-register)

$\Rightarrow$ d-consistent $x\ y\ f\ l$ = true ;

(4) (1) w-count D:Tuple $h\ d = x$:(natural, data-type) | error ;

(2) with $D\ h\ a\ f\ d\ l\ l_f = y$:(program, general-register)

$\Rightarrow$ w-consistent $x\ y\ l$ = true ;

(5) operation-consistent (unary-count O:Unary S) (unary-code $O\ r\ r'\ l\ l_f$) l
= true ;

(6) operation-consistent (binary-count O:Binary $S\ S'$)
(binary-code $O\ r'\ r''\ r\ l\ l_f$) l = true .

Proof: The definitions of 'a-count', 'u-count', 'd-count', and 'w-count' are mutually recursive, and so are the definitions of 'perform', 'unf', 'evaluate', and 'with'. We therefore need to prove (5) and (6) first, and then the conjunction of (1)—(4).

The proofs of (5) and (6) are straightforward. We will consider only one of the cases, the others are similar.

Consider the unary operation "not". The involved definitions are C.3.6.(1) and C.4.6.(1) that is

- unary-count "not" S:type =
  if S is truth-value-type then (2, truth-value-type) else error .
- unary-code "not" $r\ r'\ l\ l_f$ = overlay(
  map of sum(l,0) to (move 1 to global),
  map of sum(l,1) to (move difference global r to r')) ;

The definition of 'operation-consistent _ _ _' in E.1.1.(6) is

- operation-consistent (x : (natural, type) | error) $p\ l$ =
  if x is error then true else (well-placed $p\ l$ (component#1 of x)) .

If not 'S is truth-value-type', then the result is immediate. Otherwise, we need to prove 'well-placed (unary-code "not" $r\ r'\ l\ l_f$) l 2'. This follows immediately from the definition of 'well-placed _ _ _' in E.1.1.(1).

We then turn to the proof of the conjunction of (1)–(4). The proof is by induction on the length of inference of formulas on the form (1.1), (2.1), (3.1), and (4.1).

In the base case, we consider inferences that involve only one clause for either 'a-count', 'u-count', 'd-count', or 'w-count'. We will only give the details of one of the cases, the others are similar.

Consider the 'a-count' clause for "complete" in C.3.1.(1) that is

- a-count "complete" h d = ac-state sum(e-size,6) true () false () empty-list .

The 'perform' clause for "complete" in C.4.1.(1) is

- (1) r = free-register f
  $\Rightarrow$ perform "complete" h a f d u_n u_e u_f l $_n$ l_e l_f = a-state overlay(
  empty-list-code r l, putcommit sum(l,3) 0, finalize sum(l,6)
  u_n 0 l_n) r a ;

The definition of 'a-consistent _ _ _ _' in E.1.1.(2) is

- a-consistent (ac-state n z_n h_n z_e h_e e) (a-state p a_n a_e) u_n l = both(
  well-placed p l n,
  either(e is empty-list, u_n is 0)) .

The second part of this predicate follows immediately because e is the empty list. The first part follows from the bode macro sizes lemma and a simple calculation.

In the induction step, we consider the remaining clauses for 'a-count', 'u-count', 'd-count', and 'w-count'. We will only give the details of one of the cases, the others are similar.

Consider the 'a-count' clause for ⟦ "enact" "application" D:Dependent "to" D': Tuple ⟧ in C.3.1.(15) that is

- (1) d-count D h d = n:natural,
  abstraction-type h':data-type z_n:truth-value h_n:data-type
  z_e:truth-value h_e:data-type d':symbol-table) ;
  (2) w-count D' h d = wc-state n':natural h':data-type ;
  $\Rightarrow$ a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ h d =
  ac-state sum(n, n',27) z_n h_n z_e h_e empty-list .

The ‘perform’ clause for ⟦ “enact” “application” D:Dependent “to” D':Tuple ⟧ in C.4.1.(14) is

- (1) d-count D h d = (n:natural, abstraction-type h':data-type z_n:truth-value h_n:data-type z_e:truth-value h_e:data-type d':symbol-table) ;
  (2) w-count D' h d = wc-state n':natural h':dataatype ;
  (3) l' = sum(l, n) ;
  (4) l'' = sum(l', n') ;
  (5) evaluate D h a union(f, set of a) d l sum(l'',21) = (p:program, r:general-register) ;
  (6) with D' h a union(f, set of r) d l' sum(l'',21) = (p':program, r':general-register) ;
  ⇒ perform ⟦ “enact” “application” D:Dependent “to” D':Tuple ⟧ h a f d u_n u_e u_f l l_n l_e l_f = a-state overlay(p, p', call-sequence l'' r r' u_n u_e u_e u_f l_n l_e l_f) r r .

By applying the induction hypothesis twice and by using the code macros sizes lemma we get

- d-consistent (d-count D h d) (evaluate D h a union(f, set of a) d l sum(l'',21)) union(f, set of a) l = true ;
- w-consistent (w-count D' h d) (with D' h a union(f, set of r) d l' sum(l'',21)) l' = true ;
- well-placed (call-sequence l'' r r' u_n u_e u_f l_n l_e l_f) l'' 27 = true .

To recall, the definition of ‘a-consistent _ _ _ _’ in E.1.1.2) is

- a-consistent (ac-state n z_n h_n z_e h_e e) (a-state p a_n a_e) u_n l = both(well-placed p l n, either(e is empty-list, h is 0)) .

The second part of this predicate follows immediately because e is the empty list. The first part follows from the equations for l' and l'' and a simple calculation. □

Lemma: (Consistent use of Symbol-tables)

d : symbol-table ;
l : linenumber

$\Rightarrow$

(1) (1) find k d l = p:program
$\Rightarrow$ find-consistent (find-count k:token d) p l = true .

Proof: The lemma is proved by induction on the length of inference of (1.1). The definition of 'find _ _ _' in C.4.5.(1) is

```
•   find k:token concatenation(list of e: block, d:symbol-table) l:linenumber =
        if (block-find-count k e) is error
        then overlay( map l to ( load global 0 stack into global ),
            find k d sum(l,1))
        else empty-map .
```

The definition of 'find-consistent _ _ _' in E.1.1.(5) is

```
•   find-consistent (x : (natural, type, natural) | error) p l =
        if x is error then true else
        (well-placed p l (component#1 of x)) .
```

In the base case, we consider inferences of length one, so the rule for 'find _ _ _' has only been applied once. Hence, 'block-find-count k e' is *not* 'error' and 'p = empty-map'. In the definition of 'find-count _ _', see C.3.5, it is easy to see that 'find-count k d' then either yields 'error', (in which case the conclusion immediately follows) or yields a tuple with the first component being 0 (in which case the conclusion follows from the definition of 'well-placed _ _ _', see E.1.1.(1).

In the induction step, we have that 'block-find-count k e' is 'error' and that 'find k d sum(l,1)' is an individual. In the definition of 'find-count _ _', see C.3.5, it is easy to see that 'find-count k d' then either yields 'error', (in which case the conclusion immediately follows) or else involves the clause C.3.5.(3) that is

• (1) block-find-count k e = error ;
(2) find-count k d = (n:natural, S:type, j:natural)

$\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table) =
(sum(n,1), S, j) .

We then need to prove 'well-placed (find k d l) l sum(n,1)'. By applying the induction hypothesis to 'find k d sum(l,1)' we get that

- find-consistent (find-count k:token d) (find k d sum(l,1)) sum(l,1) = true .

Hence, 'well-placed (find k d sum(l,1)) sum(l,1) n'', from which the desired conclusion follows by a simple calculation. □

Lemma: (Increasing Type Analysis of Unfoldings)

h , h_n , h'_n , h_e , h'_e : data-type ;
z_n , z'_n , z_e , z'_e : truth-value :
d : symbol-table ;
e , e' : block ;
n' : natural

$\Rightarrow$

(1) (1) u-count U:Unf h d z_n h_n z_e h_e e = ac-state n' z'_n h'_n z'_e h'_e e'
$\Rightarrow$ ac-less z_n h_n z_e h_e e z'_n h'_n z'_e h'_e e' =true .

Proof: The lemma is proved by induction on the length of inference of (1.1).

In the base case there is only one clause to consider, namely C.3.2.(6) that is

- u-count "unfold" h d z_n h_n z_e h_e e = ac-state 2 z_n h_n z_e h_e e .

The conclusion of the lemma is immediate.

In the induction step we consider the five other clauses in C.3.2. The fifth of them is

- u-count ⟦ U:Unf "or" A:Act ⟧ h d z_n h_n z_e h_e e =
u-count ⟦ A "or" U ⟧ h d z_n h_n z_e h_e e ;

By applying the induction hypothesis, we get the conclusion of the lemma.

We give the details of only the second of the remaining four cases, the others are similar. The involved clause is

- (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e empty-list ;
  (2) compare-data-types z_e h_e z'_e h'_e = h''''_e:data-type ;
  (3) u-count U h'_n d z_n h_n either(z_e, z'_e) h'''_e e = ac-state n'' z''_n h''_n z''_e h''_e e''
  ⇒ u-count ⟦ A:Act "then" U:Unf ⟧ h d z_n h_n z_e h_e e =
  ac-state sum(n',7,n'') z''_n h''_n z''_e h''_e e'' ;

By applying the induction hypothesis to the third assumption we get that 'cc-less z_n h_n e z''_n h''_n e''' and 'ec-less either(z_e,z'_e) h'''_e z''_e h''_e' (see the definition of 'ac-less' in E.1.2). We then only need to prove 'ec-less z_e h_e z''_e h''_e'. It is easy to see that 'ec-less' is transitive, so it is sufficient to prove 'ec-less z_e h_e either(z_e,z'_e) h''''_e'. If z_e is false, then this is immediate. If z_e is true, then we also have that either(z_e,z'_e) is true. We then have to prove that 'h_e is h''''_e'. To do this we need the second assumption. The definition of 'compare-data-types _ _ _ _' in C.1.2.(8) is

- compare-data-types z h z' h' =
  if z' is false then h else
  if both(z is false, z' is true) then h' else
  if all(z is true,z' is true, h is h') then h else nothing .

It follows that because z_e is true, then 'h_e = h''''_e'. □

Lemma: (Type Analysis of Unfoldings computes a Fixed Point)

h , h_n , h'_n , h_e , h'_e : data-type ;
z_n , z'_n , z_e , z'_e : truth-value :
d : symbol-table ;
e , e' : block ;
n' : natural
⇒
(1) (1) u-count U:Unf h d z_n h_n z_e h_e e = ac-state n' z'_n h'_n z'_e h'_e e'
⇒ u-count U:Unf h d z'_n h'_n z'_e h'_e e = ac-state n' z'_n h'_n z'_e h'_e e'

Proof: The lemma is proved by induction on the length of inference of (1.1).

In the base case there is only one clause to consider, namely C.3.2.(6) that is

- u-count "unfold" h d z_n h_n z_e h_e e = ac-state 2 z_n h_n z_e h_e e .

The conclusion of the lemma is immediate.

In the induction step we consider the five other clauses in C.3.2. The fifth of them is

- $\textsf{u-count}\ [\![\ U{:}\textsf{Unf}\ \text{“}\textsf{or}\text{”}\ A{:}\textsf{Act}\ ]\!]\ h\ d\ z_n\ h_n\ z_e\ h_e\ e =$
  $\textsf{u-count}\ [\![\ A\ \text{“}\textsf{or}\text{”}\ U\ ]\!]\ h\ d\ z_n\ h_n\ z_e\ h_e\ e$;

By applying the induction hypothesis, we get the conclusion of the lemma.

We give the details of only the second of the remaining four cases, the others are similar. The involved clause is

- (1) $\textsf{a-count}\ A\ h\ d = \textsf{ac-state}\ n'\ z_n'\ h_n'\ z_e'\ h_e'\ \textsf{empty-list}$;
  (2) $\textsf{compare-data-types}\ z_e\ h_e\ z_e'\ h_e' = h_e'''{:}\textsf{data-type}$;
  (3) $\textsf{u-count}\ U\ h_n'\ d\ z_n\ h_n\ \textsf{either}(z_e, z_e')\ h_e'''\ e = \textsf{ac-state}\ n''\ z_n''\ h_n''\ z_e''\ h_e''\ e''$
  $\Rightarrow$ $\textsf{u-count}\ [\![\ A{:}\textsf{Act}\ \text{“}\textsf{then}\text{”}\ U{:}\textsf{Unf}\ ]\!]\ h\ d\ z_n\ h_n\ z_e\ h_e\ e =$
  $\textsf{ac-state}\ \textsf{sum}(n',7,n'')\ z_n''\ h_n''\ z_e''\ h_e''\ e''$;

By applying the induction hypothesis to the third assumption we get that

- $\textsf{u-count}\ U{:}\textsf{Unf}\ h_n'\ d\ z_n''\ h_n''\ z_e''\ h_e''\ e'' = \textsf{ac-state}\ n''\ z_n''\ h_n''\ z_e''\ h_e''\ e''$

It is sufficient to prove

- $\textsf{either}(z_e'', z_e') = z_e''$.
- $\textsf{compare-data-types}\ z_e''\ h_e''\ z_e'\ h_e' = h_e''$.

We will split the proof into two cases.

Consider first ‘$z_e'' = \textsf{false}$’. From the lemma on increasing type analysis of unfolding we get that

- $\textsf{ec-less}\ \textsf{either}(z_e, z_e')\ h_e'''\ z_e''\ h_e'' = \textsf{true}$.

Hence, z_e and z_e' are both false. The validity of the two formulas follows.

Consider next ‘$z_e'' = \textsf{true}$’. Here, ‘$\textsf{either}(z_e'', z_e') = z_e''$’ is immediate. If z_e' is false, then the second formula immediately follows. Consider therefore the case where z_e' is true. We need to prove ‘$h_e'' = h_e'$’. To prove this, we again use

- $\textsf{ec-less}\ \textsf{either}(z_e, z_e')\ h_e'''\ z_e''\ h_e'' = \textsf{true}$.

From this, it follows that

- $h_e'' = h_e''' =$ compare-date-types $z_e\ h_e\ z_e'\ h_e' = h_e$.

The last equality holds because the result is known in advance to be an individual. □

E.3 Correctness of Analysis

needs: Auxiliary Notation .

Lemma: (Correctness of Analysis)

h, h_n, h_n', h_e, h_e' : data-type ;
z_n, z_n', z_e, z_e' : truth-value ;
d : symbol-table ;
e, e' : block ;
n, n' : natural ;
b : bindings ;
v, v' : datum ;
S, S' : type ;

⇒

(1) (1) a-count $A\ h\ d =$ ac-state $n\ z_n\ h_n\ z_e\ h_e\ e$;
(2) tc-abs h :- t ;
(3) bc-abs d :- b ;
(4) final A:Act t:data b:bindings s:storage io:input-output = m_a:state
⇒ mc-abs (ac-state n:natural z_n:truth-value h_n:data-type
z_e:truth-value h_e:data-type e:block) :- m_a ;

(2) (1) u-count $U\ h\ d\ z_n\ h_n\ z_e\ h_e\ e =$ ac-state $n'\ z_n'\ h_n'\ z_e'\ h_e'\ e'$;
(2) u-count $U'\ h'\ d$ false () false () empty-list = ac-state $n\ z_n'\ h_n'\ z_e'\ h_e'\ e'$;
(3) tc-abs h :- t ;
(4) bc-abs d :- b ;
(5) unf-final U:Unf ⟦ "unfolding" U' ⟧
t:data b:bindings s:storage io:input-output = m_a:state

$\Rightarrow$ mc-abs (ac-state n':natural z'_n:truth-value h'_n:data-type
  z'_e:truth-value h'_e:data-type e':block) :- m_a ;

(3) (1) d-count D h d = error ;
(2) tc-abs h :- t ;
(3) bc-abs d :- b
$\Rightarrow$ evaluated D:Dependent t:data b:bindings s:storage = nothing ;

(4) (1) d-count D h d = (n:natural,S:type) ;
(2) tc-abs h :- t ;
(3) bc-abs d :- b ;
(4) evaluated D:Dependent t:data b:bindings s:storage = v:datum
$\Rightarrow$ vc-abs S :- v ;

(5) (1) w-count D h d = error ;
(2) tc-abs h :- t ;
(3) bc-abs d :- b
$\Rightarrow$ multi-evaluated D:Tuple t:data b:bindings s:storage = nothing ;

(6) (1) w-count D h d = wc-state n:natural h':data-type ;
(2) tc-abs h :- t ;
(3) bc-abs d :- b
(4) multi-evaluated D:Tuple t:data b:bindings s:storage = t':data
$\Rightarrow$ tc-abs h' :- t' ;

(7) (1) find-count k d = error ;
(2) bc-abs d :- b
$\Rightarrow$ b at k:token = nothing ;

(8) (1) find-count k d = (n:natural, S:type, j:natural) ;
(2) bc-abs d :- b
$\Rightarrow$ vc-abs S :- (b at k:token) ;

(9) (1) block-find-count k e = error ;
(2) ec-abs e :- b
$\Rightarrow$ b at k:token = nothing ;

(10) (1) block-find-count k e = (S:type, j:natural) ;
(2) ec-abs e :- b
$\Rightarrow$ vc-abs S :- (b at k:token) ;

(11) (1) unary-count O S = error ;
(2) vc-abs S :- v
$\Rightarrow$ unary-operation O:Unary v = nothing ;
(12) (1) unary-count O S = (n:natural, S':type) ;
(2) vc-abs S :- v ;
(3) unary-operation O:Unary v = v':datum
$\Rightarrow$ vc-abs S' :- v' ;
(13) (1) binary-count O S S' = error ;
(2) vc-abs S :- v ;
(3) vc-abs S' :- v'
$\Rightarrow$ binary-operation O:Binary v v' = nothing ;
(14) (1) binary-count O S S' = (n:natural, S'':type) ;
(2) vc-abs S :- v ;
(3) vc-abs S' :- v' ;
(4) binary-operation O:Binary v v' = v'':datum
$\Rightarrow$ vc-abs S'' :- v'' .

Proof: The definitions of 'a-count _ _ _', 'u-count _ _ _ _ _ _ _ _ _', 'd-count _ _ _', and 'w-count _ _ _' are mutually recursive. We therefore need to prove first (9)–(14), then (7)–(8), and finally the conjunction of (1)–(6).

The proofs of (9)–(14) are straightforward. We will consider only two of the cases, the others are similar.

Consider the unary operation "not" and the points (11) and (12). The involved definitions are C.3.6.(1) and A.3.5.(1) that is

- unary-count "not" S:type =
  if S is truth-value-type then (2, truth-value-type) else error .
- unary-operation "not" v = not v .

For the proof of (11), note that if 'unary-count "not" S = error', then S cannot be 'truth-value- type'. From the definition of 'vc-abs _' in E.1.2.(13)–(18) we then get that v cannot be included in 'truth-value'. Hence, 'unary-operation "not" v = nothing', as desired.

For the proof of (12), note that if 'unary-count "not" S = (n:natural, S':type)', then S must be 'truth-value-type', n must be 2, and S' must be 'truth-value-type'. One of the clauses that define 'vc-abs _' is E.1.2.(13) that is

- vc-abs truth-value-type = truth-value .

The operation 'not _' will yield an individual contained in 'truth-value' when applied to an individual contained in 'truth-value'. From this the conclusion follows.

Let us then consider the proof of (7)–(8). The proof is by induction in the length of inference of formulas on the form (7.1) and (8.1), since the definition of 'find-count _ _' is recursive. In the base case, we consider inferences of length one, so only one clause for 'find-count _ _' has been applied. The only two possible such clauses are C3.5.(1)–(2) that is

- find-count k:token empty-list = error .
- (1) block-find-count k e = (S:type, j:natural)

  ⇒ find-count k:token concatenation(list of e:block, d:symbol-table) =
  (0, S, j) .

In the first case, we need to prove the conclusion of (7). From the definition of 'bc-abs _' in D.2.(3)–(4) we get that b must be the empty map. From this the conclusion follows immediately. In the second case, we need to prove the conclusion of (8). From the definition of 'bc-abs _' in E.1.2.(4)–(6) we get that

- (1) ec-abs e :- b ;

  (2) bc-abs d :- b' ;

  ⇒ bc-abs concatenation(list of e:block, d:symbol-table) :- overlay(b, b') .

By using the lemma's point (10) we get that 'vc-abs S :- (b at k)'. From this it immediately follows that 'vc-abs S :- (overlay(b, b') at k)', as desired.

In the induction step, we consider the remaining two clauses for 'find-count _ _' namely C.3.5.(3)–(4) that is

(1) (1) block-find-count k e = error ;

(2) find-count k d = (n:natural, S:type, j:natural)

⇒ find-count k:token concatenation(list of e:block, d:symbol-table) =
(sum(n,1), S, j) .

(2) (1) block-find-count k e = error ;
(2) find-count k d = error
$\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table) = error .

In the first case, we need to prove the conclusion of (8). Like above, we get that

- (1) ec-abs e :- b ;
  (2) bc-abs d :- b'
  $\Rightarrow$ bc-abs concatenation(list of e:block, d:symbol-table) :- overlay(b, b') .

By using the lemma's point (9) and the induction hypothesis we get that k is not in the domain of b and that 'vc-abs S :- (b' at k)'. From this it immediately follows that 'vc-abs S :- (overlay(b, b') at k)', as desired.

In the second case, we need to prove the conclusion of (7). Similar to the first case this follows from the lemma's point (9) and the induction hypothesis.

Let us finally consider the proof of the conjunction of (1)–(6). The proof is by induction in the length of inference of formulas on the form (1.1), (2.5), (3.1), (4.1), (5.1), and (6.1).

In the base case, we consider inferences that involve only one clause for either 'a-count', 'u-count', 'd-count', or 'w-count'. We will only give the details of one of the cases, the others are similar.

Consider the 'a-count' clause for "complete" in C.3.1.(1) that is

- a-count "complete" h d = ac-state sum(e-size,6) true () false () empty-list .

The 'final' clause for "complete" in A.3.1.(1) is

- final "complete" t b s io = completed () empty-map s io uncommitted .

To prove the conclusion of (1), we need a clause for 'mc-abs' namely E.1.2(12) that is

- (1) tc-abs h_n :- t ;
  (2) ec-abs e :- b
  $\Rightarrow$ mc-abs (ac-state n:natural true h_n:data-type, z_e:truth-value h_e:data-type $e$block) :- completed t b s io c ;

From this the conclusion immediately follows.

In the induction step, we consider the remaining clauses for 'a-count', 'u-count', 'd-count', and 'w-count'. We will only give the details of four of the cases, the others are similar.

Firstly, consider the 'a-count' and 'final' clauses for ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ namely C.3.1.(15) and A.3.1.(15) that is

- (1) d-count D h d = (n:natural,
  abstraction-type h':data-type z_n:truth-value h_n:data-type
  z_e:truth-value h_e:data-type d':symbol-table) ;
  (2) w-count D' h d = wc-state n':natural h':data-type ;
  ⇒ a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ h d =
  ac-state sum(n,n',27) z_n h_n z_e h_e empty-list .
- (1) evaluated D t b s = closure-abstraction A:Act D'':Data b':bindings ;
  (2) multi-evaluated D' t b s = t':data
  ⇒ final ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ t b s io =
  final A t' b' s io ;

By applying the induction hypothesis twice, points (4) and (6), we get

- vc-abs (abstraction-type h':data-type z_n:truth-value h_n:data-type
  z_e:truth-value h_e:data-type d':symbol-table) :-
  closure-abstraction A:Act D'':Data b':bindings .
- tc-abs h' :- t' .

We then need one of the defining clauses for 'vc-abs _' namely E.1.2.(17), and using that we can further assume that

- a-count A h' concatenation(list of empty-list, d') =
  ac-state n:natural z_n h_n z_e h_e empty-list .
- data-type D = h' .
- bc-abs d' :- b' .

We can now apply the induction hypothesis, point (1), and that immediately gives the conclusion.

Secondly, consider the 'a-count' and 'final' clauses for ⟦ "unfolding" U ⟧ namely C.3.1.(17) and A.3.1.(17) that is

- (1) u-count U h d false () false () empty-list = ac-state n z_n h_n z_e h_e e

  $\Rightarrow$ a-count ⟦ "unfolding" U:Unf ⟧ h d = ac-state sum (4,n,18) z_n h_n z_e h_e e ;
- final ⟦ "unfolding" U:Unf ⟧ t b s io = unf-final U ⟦ "unfolding" U ⟧ t b s io ;

To prove the conclusion of (1), we use these two clauses to bring us in a situation where we can apply the induction hypothesis, point (2). This immediately yields the desired result, since the 'natural' component of an 'ac-state' has no impact on what is yielded by 'mc-abs'.

Thirdly, consider the 'a-count' clause and the first 'final' clause for ⟦ A:Act "then" A':Act ⟧ namely C.3.1.(19) and A.3.1.(21) that is

- (1) a-count A h d = ac-state n' z'_n h'_n z'_e h'_e empty-list ;

  (2) a-count A' h'_n d = ac-state n'' z''_n h''_n z''_e h''_e e ;

  (3) compare-data-types z'_e h'_e z''_e h''_e = h_e:data-type

  $\Rightarrow$ a-count ⟦ A:Act "then" A':Act ⟧ h d = ac-state sum(n',2,n'',18) both(z'_n,z''_n) h''_n) h_e e .
- (1) final A t b s io = completed t' empty-map s' io' c' ;

  (2) final A' t' b s' io' = completed t'' b'' s'' io'' c''

  $\Rightarrow$ final ⟦ A:Act "then" A':Act ⟧ t b s io := completed t'' b'' s'' io'' either(c',c'') .

By applying the induction hypothesis, point (1), to the first assumptions we get

- mc-abs (ac-state n' z'_n h'_n z'_e h'_e empty-list) :-
  completed t' empty-map s' io' c' .

From one of the clauses for 'mc-abs' E.1.2.(12) we get that we further can assume 'tc-abs h'_n :- t'', and that z'_n must be true. We can then apply the induction hypothesis, point (1), this time to the second assumptions, and we get

- mc-abs (ac-state n'' z''_n h''_n z''_e h''_e e) :-
  completed t'' b'' s'' io'' c'' .

From one of the clauses for 'mc-abs' E.1.2.(12) we get that we further can assume 'tc-abs h''_n :- t''' and 'ec-abs e :- b''' and z''_n must be true.

To prove the conclusion of point (1) , we note that we have the established the explicit assumptions of the clause E.1.2.(12) for 'mc-abs'. We then only need to prove that 'both(z'_n,z''_n)' is true. This follows immediately because both components are true.

Fourthly, consider the 'u-count' clause for "unfold" in C.3.2.(6) that is

- u-count "unfold" $h\ d\ z_n\ h_n\ z_e\ h_e\ e$= ac-state 2 $z_n\ h_n\ z_e\ h_e\ e$.

The 'unf-final' clause for "unfold" in A.3.2.(13) and the 'final' clause for ⟦ "unfolding" U ⟧ in A.3.1.(17) are

- unf-final "unfold" ⟦ unfolding U:Unf ⟧ $t\ b\ s\ io$ = final ⟦ "unfolding" U ⟧ t $b\ s\ io$.
- final "unfolding" U:Unf ⟧ $t\ b\ s\ io$ = unf-final U ⟦ "unfolding" U ⟧ $t\ b\ s$ io ;

To prove the conclusion of (2), we use these two clauses to bring us in a situation where we can apply the induction hypothesis, point (2). This immediately yields the desired result, since the 'natural' component of an 'ac-state' has no impact on what is yielded by 'mc-a bs'. □

E.4 Completeness

needs: **Auxiliary Notation ,**
Compiler Consistency ,
Correctness of Analysis .

Lemma: **(Sound Semantics of Types)**

n : natural ;
q : memory ;
p : program ;
h, h_n, h_e, : data-type ;
t : data ;
d : symbol-table ;

b : bindings ;
e : block ;
m_p : sparc-state ;
z_n, z_e : truth-value ;
a_n, a_e : general-register ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
m_a : state ;
i : integer ;
S : type ;
v: datum ;

$\Rightarrow$

(1) (1) t-abs $n\ q\ p\ h$:- t
$\Rightarrow$ tc-abs h :- t ;
(2) (1) b-abs $n\ q\ p\ d$:- b
$\Rightarrow$ bc-abs d :- b ;
(3) (1) e-abs $n\ q\ p\ e$:- b
$\Rightarrow$ ec-abs e :- b ;
(4) (1) m-abs $m_p\ z_n\ h_n\ z_e\ h_e\ a_n\ a_e\ e\ l_n\ l_e\ l_f$:- m_a
$\Rightarrow$ mc-abs (ac-state $n\ z_n\ h_n\ z_e\ h_e\ e$) :- m_a ;
(5) (1) v-abs $i\ q\ p\ S$:- v
$\Rightarrow$ vc-abs S :- v .

Proof: The definitions of 'b-abs _ _ _', 'e-abs _ _ _', and 'v-abs _ _ _ _' are mutually recursive, and so are the definitions of 'bc-abs _ _ _ _', 'ec-abs _ _ _ _', and 'vc-abs _ _ _ _'. We therefore need to prove the conjunction of (2), (3), and (5) before the others. After that we must prove (1) and finally we must prove (4). These three subproofs are similar, however, so here we give a detailed proof of only the conjunction of (2), (3), and (5).

The proof is by induction on the length of inference of formulas on the form (2.1), (3.1), and (5.1). We will only show the treatment of the clauses for the defining of 'v-abs', in D.2.(16)–(23), and 'vc-abs' in E.1.2.(13)–(18). The clauses for 'b-abs', 'bc-abs', 'e-abs', and 'ec-abs' can be treated similarly.

In the base case, we consider the clauses for the four base types and the abstractiontype constructor. Each of the cases involving a base type are similar. Let us here look at only the first, the one concerning 'truth-value-type'. The involved definitions are D.2.(16)–(17) and E.1.2.(13) that is

- v-abs 0 q p truth-value-type :- false ;
- v-abs 1 q p truth-value-type :- true ;
- vc-abs truth-value-type = truth-value .

The conclusion is then immediate because 'false,true:truth-value'.

In the induction step, we consider the clauses for the abstraction-type constructor and the list-type constructor.

The case with the abstraction-type constructor involves the clauses D.2.(21) and E.1.2.(17) that is

- (1) a-count A h concatenation(list of empty-list, d) =
  ac-state n z_n h_n z_e h_e empty-list ;
  (2) l = (q at heap) at i ;
  (3) l' = sum(l, n) ;
  (4) perform A h (reg 0) empty-set concatenation(list of empty-list, d) 0 0 0
  l l' sum(l',2) sum(l',3) = a-state p' $_n$ a_e ;
  (5) return-sequence a_n a_e l' = p'' ;
  (6) p' is submap of p = true ;
  (7) p'' is submap of p = true ;
  (8) data-type D = h ;
  (9) b-abs ((q at heap) at sum(i,1)) q p d :- b
  $\Rightarrow$ v-abs i:integer q p (abstraction-type h z_n h_n z_e h_e d) :-
  closure-abstraction A h concatenation(list of empty-list, d) =
- (1) a-count A h concatenation(list of empty-list, d) =
  ac-state n:natural z_n h_n z_e h_e empty-list ;
  (2) data-type D = h ;
  (3) bc-abs d :- b
  $\Rightarrow$ vc-abs (abstraction-type h:data-type z_n:truth-value h_n:data-type
  z_e:truth-value h_e:data-type d:symbol-table) :-
  closure-abstraction A:Act D:Data b:bindings .

Assume the conclusion of the first of these clauses. We can then also assume the nine assumptions of that clause. To prove the lemma we need to prove the three assumptions of the second clause. The first two of these are immediate, the third follows by using the induction hypothesis point (2).

The case with the list-type constructor involves the clauses D.2.(22)–(23) and E.1.2.(18) that is

- (1) ((*q* at heap) at sum(*i*,1)) is -1 = true
  ⇒ v-abs *i*:integer *q p* (list-type *S*) :- empty-list ;
- (1) v-abs ((*q* at heap) at *i*) *q p S* :- *v* ;
  (2) v-abs ((*q* at heap) at sum(*i*,1)) *q p* list-type *S*:type) :- *v*′ ;
  ⇒ v-abs *i*:integer *q p* (list-type *S*) :- concatenation(list of *v*, *v*′) ;
- vc-abs (list-type *S*:type) = [vc-abs *S*] list .

Firstly, assume the conclusion of the first clause. Since '*S*' is an individual, we immediately get '[vc-abs *S*] list :- empty-list'. Secondly, assume the conclusion of the second clause. We can then also assume the two assumptions of that clause. We need to prove 'vc-abs *S* :- *v*' and 'vc-abs (list-type *S*) :- *v*′'. Both follow from the induction hypothesis. □

Lemma: (Singlethreadedness of Actions)

t : data ;
b : bindings ;
s : storage ;
io : input-output ;

⇒

(1) (1) final *A* Act *t b s io* = $m_a$state
⇒ either(both((storage of m_a) is *s*, (input-output of m_a) is *io*),
(commitment of m_a) is committed) = true ;

(2) (1) unf-final *U*:Unf ⟦ "unfolding" *U*′:Unf ⟧ *t b s io* = m_a:state
⇒ either(both((storage of m_a) is *s*, (input-output of m_a) is *io*),
(commitment of m_a) is committed) = true .

Proof: By induction in the length of inference. See section 4.2.2 for the proof. □

Lemma: (Completeness)

g globals ;
w : windows ;
n, n', n'', nw : natural ;
q, q', q'' : memory ;
p, p', p'' : program ;
h, h', h_n, h'_n, h_e, h'_e : data-type ;
t : data ;
d : symbol-table ;
b : bindings ;
e, e's : block ;
m_p, m'_p : sparc-state ;
cz : was-zero ;
cn : was-negative ;
z_n, z'_n, z_e, z'_e : truth-value ;
r, r', r'', a, a', a_n, a'_n, a_e, a'_e : general-register ;
l linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
l_u :linenumber-unfold ;
S, S' : type ;
v, v' : datum ;
io :input-output ;
s : storage ;
f, f' : frosen ;
u_n, u_e, u_f :cleanup $\Rightarrow$

(1) (1) a-count A h d = ac-state n' z_n h_n z_e h_e e ;
(2) perform A h a f d u_n u_e u_f l l_n l_e l_f = a-state p' a_n a_e ;
(3) t-abs ((head of w) at a) q' p h :- t ;
(4) b-abs((head of w) at statlink q' p d :- b ;
(5) store-abs (g at firstfree)(q at store) :- s ;
(6) io:input-output = (il:[integer] list, ol:[integer] list) ;
(7) i-abs ((q at input) at 0)(q at input) = il ;

(8) o-abs ((*q* at output) at 0)(*q* at output) = ol ;

(9) pre-condition (p' p sum(l, n')(l_n, l_e, l_f) g w q q' nw d = true ;

(10) final A:Act t, b s io = m_a:state

$\Rightarrow$ $\exists m_p$:sparc-state $\exists n$:natural

(11) spare-final n (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) nw = m_p ;

(12) a-post-condition g w q m_p f u_n u_e u_f e (commitment of m_a) = true ;

(13) (1) m-earlier m_p m'_p = true

$\Rightarrow$ m-abs m'_p z_n h_n z_e h_e a_n a_e e l_n l_e l_f :- m_a ;

(2) (1) u-count U h d z_n h_n z_e h_e e = ac-state n' z'_n h'_n z'_e h'_e e' ;

(2) u-count U' h' d false () false () empty-list = ac-state n'' z'_n h'_n z'_e h'_e e' ;

(3) unf U h a f d u_n u_e u_f a' z_n h_n a_n z_e h_e a_e e l l_n l_e l_f l_u = a-state p' a'_n a'_e ;

(4) unf U' h' a' f' d u_n u_e u_f a' false () a' false () a' empty-list l_u l_n l_e l_f l_u = a-state p'' a'_n a'_e ;

(5) t-abs ((head of w) at a) q' p h :- t ;

(6) b-abs ((head of w) at staticlink) q' p d :- b ;

(7) store-abs (g at firstfree) (q at store) :- s ;

(8) io:input-output = (il:[integer] list, ol:[integer] list) ;

(9) i-abs ((q at input) at 0) (q at input) = il ;

(10) o-abs ((q at output) at 0) (q at output) = ol ;

(11) precondition p' p sum(l,n') (l_n, l_e, l_f) g w q q' nw d = true ;

(12) p'' is submap of p = true ;

(13) sum(l_u, n'') leq (l_n, l_e, l_f) = true ;

(14) a' is in f = false ;

(15) unf-final U:Unf $\llbracket$ "unfolding" U':Unf $\rrbracket$ t b s io = m_a:state

$\Rightarrow$ $\exists$ m_p:sparc-state $\exists$ n:natural

(16) spare-final n (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) nw = m_p ;

(17) u-post-condition g w q m_p f u_n u_e u_f e (commitment of m_a) = true ;

(18) (1) m-earlier m_p m'_p = true

$\Rightarrow$ m-abs m'_p z'_n h'_n z'_e h'_e a'_n a'_e e' l_n l_e l_f :- m_a ;

(3) (1) d-count D h d = (n':natural, S:type) ;

(2) evaluate D h a f d l l_f = (p':program, r:general-register) ;

```
        (3)   t-abs ((head of $w$) at $a$) $q'$ $p$ $h$ :- $t$ ;
        (4)   b-abs ((head of $w$) at staticlink) $q'$ $p$ $d$ :- $b$ ;
        (5)   store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
        (6)   pre-condition $p'$ $p$ sum(l,$n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
        (7)   evaluated $D$:Dependent $t$ $b$ $s$ = $v$:datum
        $\Rightarrow$   $\exists$ $m_p$:sparc-state $\exists$ $n$:natural
        (8)   spare-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l$,$n'$) $nw$ = $m_p$ ;
        (9)   d-post-condition $g$ $w$ $q$ $m_p$ $f$ = true ;
        (10)  (1)   mq-earlier $m_p$ $q''$ = true
              $\Rightarrow$   v-abs ((head of (windows of $m_p$)) at $r$) $q''$ $p$ $S$ :- $v$ ;
(4)     (1)   d-count $D$ $h$ $d$ = ($n'$:natural, $S$:type) ;
        (2)   evaluate $D$ $h$ $a$ $f$ $d$ $l$ $l_f$ = ($p'$:program, $r$:general-register) ;
        (3)   t-abs ((head of $w$) at $a$) $q'$ $p$ $h$ :- $t$ ;
        (4)   b-abs ((head of $w$) at staticlink) $q'$ $p$ $d$ :- $b$ ;
        (5)   store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
        (6)   pre-condition $p'$ $p$ sum($l$,$n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
        (7)   evaluated $D$:Dependent $t$ $b$ $s$ = nothing
        $\Rightarrow$   $\exists$ $m_p$:sparc-state $\exists$ $n$:natural
        (8)   spare-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l_f$ $nw$ = $m_p$ ;
        (9)   d-post-condition $g$ $w$ $q$ $m_p$ $f$ = true ;
(5)     (1)   w-count $D$ $h$ $d$ = wc-state $n'$:natural $h'$:data-type ;
        (2)   with $D$ $h$ $a$ $f$ $d$ $l$ $l_f$ = ($p'$:program, $r$:general-register) ;
        (3)   t-abs ((head of $w$) at $a$) $q'$ $p$ $h$ :- $t$ ;
        (4)   b-abs ((head of $w$) at staticlink) $q'$ $p$ $d$ :- $b$ ;
        (5)   store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
        (6)   pre-condition $p'$ $p$ sum($l, n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
        (7)   multi-evaluated $D$:Tuple $t$ $b$ $s$ = $t'$:data
        $\Rightarrow$   $\exists$ $m_p$:sparc-state $\exists$ $n$:natural
        (8)   spare-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l$,$n'$) $nw$ = $m_p$ ;
        (9)   d-post-condition $g$ $w$ $q$ $m_p$ $f$ = true ;
        (10)  (1)   mq-earlier $m_p$ $q''$ = true
```

```
            ⇒   t-abs((head of (windows of $m_p$)) at $r$) $q''$ $p$ $h'$ :- $t'$ ;
(6)   (1)   w-count $D$ $h$ $d$ = wc-state $n'$:natural $h'$:data-type ;
      (2)   with $D$ $h$ $a$ $f$ $d$ $l$ $l_f$ = ($p'$:program, $r$:general-register) ;
      (3)   t-abs ((head of $w$) at $a$) $q'$ $p$ $h$ :- $t$ ;
      (4)   b-abs ((head of $w$) at staticlink) $q'$ $p$ $d$ :- $b$ ;
      (5)   store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
      (6)   pre-condition $p'$ $p$ sum($l$,$n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
      (7)   multi-evaluated $D$:Tuple $t$ $b$ $s$ = nothing
      ⇒     ∃ $m_p$:sparc-state ∃ $n$:natural
      (8)   spare-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l_f$ $nw$ = $m_p$ ;
      (9)   d-post-condition $g$ $w$ $q$ $m_p$ $f$ = true ;
(7)   (1)   find-count $k$ $d$ = ($n'$:natural, $S$:type, $j$:natural) ;
      (2)   find $k$ $d$ $l$ = $p'$:program ;
      (3)   b-abs ($g$ at global) $q'$ $p$ $S$ :- $b$ ;
      (4)   basic-pre-condition $p'$ $p$ sum($l, n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ = true
      ⇒     ∃ $m_p$:sparc-state ∃ $n$:natural
      (5)   spare-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l, n'$) $nw$ = $m_p$ ;
      (6)   f-post-condition $g$ $w$ $q$ $m_p$ = true ;
      (7)   (1)   mq-earlier $m_p$ $q''$ = true
            ⇒     v-abs sum((globals of $m_p$ at globals, $j$) $q''$ $p$ $S$ :- ($b$ at $k$:token) ;
(8)   (1)   block-find-count $k$ $e$ = ($S$:type, $j$:natural) ;
      (2)   e-abs sum($g$ at global, count of items of $e$) $q$ $p$ $e$ :- $b$
      ⇒
      (3)   (1)   (($q$ at heap) restricted to up-to ($q$ at hp)) is
                        (($q'$ at heap) restricted to up-to ($q$ at hp)) = true
            ⇒     v-abs sum($g$ at global, $j$) $q'$ $p$ $S$ :- ($b$ at $k$:token) ;
(9)   (1)   unary-count $O$ $S$ = ($n'$:natural, $S'$:type) ;
      (2)   unary-code $O$ $r$ $r'$ $l$ $l_f$ = $p'$:program ;
      (3)   v-abs ((head of $w$) at $r$) $q'$ $p$ $S$ :- $v$ ;
      (4)   pre-condition $p'$ $p$ sum($l, n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
      (5)   unary-operation $O$:Unary $v$ = $v'$:datum
      ⇒     ∃ $m_p$:sparc-state ∃ $n$:natural
```

(6) sparc-final n $(p,\, l,\, cz,\, cn,\, g,\, w,\, q)$ sum(l, n') n $w = m_p$;
(7) op-post-condition g w q m_p $r' =$ true ;
(8) (1) mq-earlier m_p $q'' =$ true
$\Rightarrow$ v-abs ((head of (windows of m_p)) at r) q'' p S' :- v' ;

(10) (1) unary-count O $S = (n'$:natural, S':type) ;
(2) unary-code O r r' l $l_f = p'$:program ;
(3) v-abs ((head of w) at r) q' p S :- v ;
(4) pre-condition p' p sum(l, n') l_f g w q q' nw $d =$ true ;
(5) unary-operation O:Unary $v =$ nothing
$\Rightarrow$ $\exists$ m_p:sparc-state $\exists$ n:natural
(6) spare-final n $(p,\, l,\, cz,\, cn,\, g,\, w,\, q)$ l_f $nw = m_p$;
(7) op-post-condition g w q m_p $r' =$ true ;

(11) (1) binary-count O S $S' = (n'$:natural, S'':type) ;
(2) binary-code O r r' r'' l $l_f = p'$:program ;
(3) v-abs ((head of w) at r) q' p S :- v ;
(4) v-abs ((head of w) at r') q' p S' :- v ;
(5) pre-condition p' p sum(l, n') l_f g w q q' nw $d =$ true ;
(6) binary-operation O:Binary v $v' = v''$:datum
$\Rightarrow$ $\exists$ m_p:sparc-state $\exists$ n:natural
(7) spare-final n $(p,\, l,\, cz,\, cn,\, g,\, w,\, q)$ sum(l, n') $nw = m_p$;
(8) op-post-condition g w q m_p $r'' =$ true ;
(9) (1) mq-earlier m_p $q'' =$ true
$\Rightarrow$ v-abs ((head of (windows of m_p)) at r'') q'' p S' :- v'' ;

(12) (1) binary-count O S $S' = (n'$:natural, S'':type) ;
(2) binary-code O r r' r'' l $l_f = p'$:program ;
(3) v-abs ((head of w) at r) q' p S :- v ;
(4) v-abs ((head of w) at r') q' p S' :- v ;
(5) pre-condition p' p sum(l, n') l_f g w q q' nw $d =$ true ;
(6) binary-operation O:Binary v $v' =$ nothing
$\Rightarrow$ $\exists$ m_p:sparc-state $\exists$ n:natural
(7) spare-final n $(p,\, l,\, cz,\, cn,\, g,\, w,\, q)$ l_f $nw = m_p$;

(8) op-post-condition $g\ w\ q\ m_p\ r'' =$ true .

Proof: The definitions of 'a-count', 'u-count', 'd-count', 'w-count' are mutually recursive, and so are the definitions of 'perform', 'unf', 'evaluate', 'with'. We therefore need to prove first (8)—(12), then (7), and anally the conjunction of (1)—(6).

Let us first consider the proofs of (8)—(12). We will consider only one of the cases, the others are similar.

Consider the unary operation "not" and the lemma's point (9). The involved definitions are C.3.6.(1), C.4.6.(1), and A.3.5.(1) that is

- unary-count "not" S:type =
  if S is truth-value-type then (2, truth-value-type) else error .
- unary-code "not" $r\ r'\ l\ l_f =$ overlay(
  map of sum(l,0) to (move 1 to global),
  map of sum(l,1) to (move difference global r to r')) .
- unary-operation "not" $v =$ not v .

First note that if 'unary-count "not" $S = (n'$:natural, S':type)', then S must be 'truth-value-type', n must be 2, and S' must be 'truth-value-type'. Two of the clauses that define 'v-abs _' are D.2.(16)–(17) that is

- v-abs 0 $q\ p$ truth-value-type :- false .
- v-abs 1 $q\ p$ truth-value-type :- true .

Let us consider the first of these two cases, the second is similar. We then have that '$v =$ false', and, using the definition of 'unary-operation "not" v', we also have '$v' =$ true'. Furthermore, we have that '(head of w) at $r = 0$'. To prove the lemma's conclusions, we choose '$n = 2$' and $m_p = (p$, sum(l,2), cz, cn, overlay(map of global to 1, g), update w (map of r to 1), q). Consider the definition of 'sparc-final _ _ _ _' in B.3.1.(1) that is

- sparc-final n:natural m_p:sparc-state lt:linenumber* nw:natural =

```
if both($n$ is 0, finished $m_p$ $lt$ $nw$)
then $m_p$
else if not any($n$ is 0, finished $m_p$ $lt$ $nw$, out-of-bound $m_p$)
    then spare-final predecessor($n$) step($m_p$) $lt$ $nw$
    else nothing .
```

We can then prove the first conclusion of the lemma by applying this clause three times, as follows. First note that n is different from 0, that 'finished m_p lt nw' is false, and that 'out-of-bound m_p' is also false. Hence, that application of the rule just given yields

- spare-final 2 (p, l, cz, cn, g, w, q) sum(l,2) nw
  = sparc-final 1 step(p, l, cz, cn, g, w, q) sum(l,2) nw
  = sparc-final 1 (p, sum(l,1), cz, cn, overlay(map of global to 1, g), w, q) sum(l,2) nw

In the second step, we used the clause for 'move' namely B.3.2.(13.2). Again, we get that 1 is different from 0, that 'finished m_p lt nw' is false, and that 'out-of-bound m_p' is also false. We can then continue the calculation as follows

- spare-final 2 (p, l, cz, cn, g, w, q) sum(l,2) nw
  = spare-final 0 step(p, sum(l,1), cz, cn, overlay(map of global to 1, g), w, q) sum(l,2) nw
  = sparcfinal 0 m_p sum(l,2) nw
  = m_p

In the second step, we used the clause for 'move' namely B.3.2.(13.3). In the third step, we used that the number of steps to execute finally reached 0, and that 'finished m_p lt nw' now is true.

To prove the second conclusion of the lemma, we note that r' is the only general-register that has been modified, that the register-windows are otherwise unchanged, that 'global' is the only global register that has been modified, and that the memory is unchanged.

To prove the third conclusion of the lemma, we note that 'v-abs 1 q p truth-value-type :- true'. We need to prove 'v-abs 1 q'' p truth-value-type :- true'. This follows immediately from the assumption 'mq-earlier m_p q'' = true'.

Let us then consider the proof of the lemma's point (7). The involved definitions are C3.5.(2)—(3) and C4.5.(1).

- (1) block-find-count k e = (S:type, j:natural)
  ⇒ find-count k:token concatenation(list of e:block, d:symbol-table) = (0, S, j) .

- (1) block-find-count k e = error ;
  (2) find-count k d = (n:natural, S:type, j:natural)
  $\Rightarrow$ find-count k:token concatenation(list of e:block, d:symbol-table) =
  (sum(n,1), S, j) .
- find k:token concatenation(list of e:block, d:symbol-table) l:linenumber =

```
if (block-end-count $k$ $e$) is error
then overlay( map $l$ to ( load global 0 stack in to global ),
      find $k$ $d$ sum($l$,1))
else empty-map .
```

The proof is by induction on the length of inference of (7.1), since the definitions of both 'find-count _ _' and 'find _ _ _' are recursive.

In the base case, we consider inferences of length one, so only one clause for 'find-count' has been applied. The only possible such clause is the first of the above. From the third of the clauses, we see that 'p' = empty-map'. We then choose '$m_p = (p, l, cz, cn, g, w, q)$' and '$n = 0$'. The first of the lemma's conclusions follows immediately, and so does the second, since nothing has been changed. To prove the third conclusion, we use one of the clauses for 'b-abs _ _ _ _' to bring us in a position where we can apply the lemma's point (8). The conclusion follows immediately from this.

In the induction step, we consider the second clause for 'find-count _ _', see above. From the third of the above clauses we get that

- p' = overlay(map l to (load global 0 stack in to global),
  find k d sum(l,1))

Executing the first instruction yields a state where we can apply the induction hypothesis. This allows us to choose m_p and n, such that

- sparc-final n (p, l, cz, cn, g, w, q) sum(l,n') nw = m_p ;
- f-post-condition g w q m_p = true ;
- (1) mq-earlier m_p q'' = true
  $\Rightarrow$ v-abs sum((globals of m_p) at globals, j) q'' p S :- (b at k:token) ;

To prove the lemma we then choose the same m_p as the final machine state, and we choose sum(n,1) as the number of steps to be executed. The conclusion follows.

Let us finally consider the proof of the conjunction of (1)—(6). The proof is by induction in the length of inference of formulas on the form (1.1), (2.15), (3.1), (4.1), (5.1), and (6.1).

In the base case, we consider inferences that involve only one clause for either 'a-count', 'u-count', 'd-count', or 'w-count' . We will only give the details of one of the cases, the others are similar.

Consider the 'a-count', 'perform', and 'final' clauses for "complete" namely C.3.1.(1), C.4.1.(1), and A.3.1.(1) that is

- a-count "complete" h d = ac-state sum(e-size,6) true () false () empty-list .
- (1) r = free-register f
  $\Rightarrow$ perform "complete" h a f d u_n u_e u_f l l_n l_e l_f = a-state overlay(
  empty-list-code r l, putcommit sum(l,3) 0, finalize sum(l,6)
  u_n 0 l_n) r a ;
- final "complete" t b s io = completed () empty-map s io uncommitted .

We choose 'n = sum(e=size,6) = 10' and

- m_p = (p, l_n, cz, cn,
  overlay(map of global to 0, map of hp to sum(g at hp,2),
  map of cp to sum(g at cp,1),
  map of sp to difference(g at sp, u_n),
  map of cef to 0, g),
  update w (map of r to (g at hp)),
  overlay(map of heap to overlay(map of sum(g at hp,1) to -1, q at heap),
  map of commits to overlay(map of g at cp to 0, q at commits),
  q))

We will then show the first conclusion of the lemma, namely that

- sparc-final 10 (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) nw = m_p

The proof will use the clause for 'sparc-final _ _ _ _' in B.3.1.(1) and the various clauses for 'step _' in B.3.2. We do not comment on the individual steps.

- sparc-final 10 (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) nw
  = sparc-final 9 (p, sum(l,1), cz, cn, overlay(map of global to -1, g), w,

```
        $q$) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 8 ($p$, sum($l$,2), $cz$, $cn$, overlay(map of global to -1, $g$), $w$,
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 7 ($p$, sum($l$,3), $cz$, $cn$, overlay(map of global to -1, $g$), $w$,
    update $w$ (map of $r$ to ($g$ at hp)), $q$ )
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 6 ($p$, sum($l$,4), $cz$, $cn$,
    overlay(map of global to -1, map of hp to sum($g$ at hp,2), $g$),
    update $w$ (map of $r$ to ($g$ at hp)),
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 5 ($p$, sum($l$,5), $cz$, $cn$,
    overlay(map of global to 0, map of hp to sum($g$ at hp,2), $g$),
    update $w$ (map of $r$ to ($g$ at hp)),
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 4 ($p$, sum($l$,6), $cz$, $cn$,
    overlay(map of global to 0, map of hp to sum($g$ at hp,2), $g$),
    update $w$ (map of $r$ to ($g$ at hp)),
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        map of commits to overlay(map of $g$ at cp to 0, $q$ at commits),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 3 ($p$, sum($l$,7), $cz$, $cn$,
    overlay(map of global to 0, map of hp to sum($g$ at hp,2),
        map of cp to sum($g$ at cp,1), $g$),
    update $w$ (map of $r$ to ($g$ at hp)),
    overlay(map of heap to overlay(map of sum($g$ at hp,1) to -1, $q$ at heap),
        map of commits to overlay(map of $g$ at cp to 0, $q$ at commits),
        $q$ ) ($l_n$, $l_e$, $l_f$) $nw$
= sparc-final 2 ($p$, sum($l$,8), $cz$, $cn$,
    overlay(map of global to 0, map of hp to sum($g$ at hp,2),
        map of cp to sum($g$ at cp,1),
        map of sp to difference($g$ at sp,$u_n$), $g$),
    update $w$ (map of $r$ to ($g$ at hp)),
```

overlay(map of heap to overlay(map of sum(g at hp,1) to -1, q at heap),
map of commits to overlay(map of g at cp to 0, q at commits),
q) (l_n, l_e, l_f) nw

= sparc-final 1 (p, sum(l,9), cz, cn,
overlay(map of global to 0, map of hp to sum(g at hp,2),
map of cp to sum(g at cp,1),
map of sp to difference(g at sp,u_n),
map of cef to 0, g),
update w (map of r to (g at hp)),
overlay(map of heap to overlay(map of sum(g at hp,1) to -1, q at heap),
map of commits to overlay(map of g at cp to 0, q at commits),
q) (l_n, l_e, l_f) nw

= sparc-final 0 (p l_n, cz, cn,
overlay(map of global to 0, map of hp to sum(g at hp,2),
map of cp to sum(g at cp,1),
map of sp to difference(g at sp,u_n),
map of cef to 0, g),
update w (map of r to (g at hp)),
overlay(map of heap to overlay(map of sum(g at hp,1) to -1, q at heap),
map of commits to overlay(map of g at cp to 0, q {textsfat commits),
q) (l_n, l_e, l_f) nw

= (p l_n, cz, cn,
overlay(map of global to 0, map of hp to sum(g at hp,2),
map of cp to sum(g at cp,1),
map of sp to difference(g at sp,u_n),
map of cef to 0, g),
update w (map of r to (g at hp)),
overlay(map of heap to overlay(map of sum(g at hp,1) to -1, q at heap),
map of commits to overlay(map of g at cp to 0, q at commits),
q))

= m_p

Most of the conjuncts in the second conclusion of the lemma are immediately satisfied. Only the following condition needs the assumption 'r = free-register f' and the calculation of free registers lemma.

- ((head of w) restricted to f) is ((head of windows of m_p) restricted to f) .

The third conclusion of the lemma follows immediately from one of the clauses for 'm-abs' namely D.2.15.(5).

In the induction step, we consider the remaining clauses for 'a-count', 'u-count', 'd-count', and 'w-count'. We will only give the details of four of the cases, the others are similar.

Firstly, consider the 'a-count', 'perform', and 'final' clauses for ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ in C.3.1.(15), C.4.1.(14), and A.3.1.(15).

- (1) d-count D h d = (n:natural,
  abstraction-type h':data-type z_n:truth-value h_n:data-type
  z_e:truth-value h_e:data-type d':symbol-table) ;
  (2) w-count D' h d = wc-state n':natural h':data-type ;
  ⇒ a-count ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ h d =
  ac-state sum(n, n',27) z_n h_n z_e h_e empty-list .
- (1) d-count D h d = (n:natural, abstraction-type h':data-type z_n:truth-value
  h_n:data-type z_e:truth-value h_e:data-type d':symbol-table) ;
  (2) w-count D' h d = wc-state n':natural h':data-type ;
  (3) l' = sum(l, n) ;
  (4) l'' = sum(l, n') ;
  (5) evaluate D h a union(f, set of a) d l sum(l'',21) =
  (p:program, r:general-register) ;
  (6) with D' h a union(f, set of r) d l' sum(l'',21) =
  (p':program, r':general-register) ;
  ⇒ perform ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧
  h a f d u_n u_e u_f l l_n l_e l_f =
  a-state overlay(p, p', call-sequence l'' r r' u_n u_e u_f l_n l_e l_f) r r .
- (1) evaluated D t b s = closure-abstraction A:Act D'':Data b':bindings ;
  (2) multi-evaluated D' t b s = t' : data
  ⇒ final ⟦ "enact" "application" D:Dependent "to" D':Tuple ⟧ t b s io =
  final A t' b' s io .

By applying the induction hypothesis point (3) we get 'm_p:sparc-state' and 'j:natural' such that

- spare-final j (prg, l, cz, cn, g, w, q) sum(l,n) nw = m_p ;

- d-post-condition g w q m_p union(f, set of a) = true ;
- (1) mq-earlier m_p q'' = true
  $\Rightarrow$ v-abs ((head of (windows of m_p)) at r) q'' prg
  (abstraction-type h':data-type z_n:truth-value
  h_n:data-type z_e:truth-value h_e:data-type d':symbol-table) :-
  (closure-abstraction A:Act D'':Data b':bindings) ;

Note the variable prg, it is the one which in the lemma is called p.

We can then apply the induction hypothesis point (5) and get 'm'_p:sparc-state' and 'j':natural' such that, after a simplification,

- sparc-final sum(j, j') (prg, l, cz, cn, g, w, q) sum(l,n,n') nw = m'_p ;
- d-post-condition g w q m'_p union(f, set of (a, r)) = true ;
- (1) mq-earlier m'_p q'' = true
  $\Rightarrow$ t-abs ((head of (windows of m'_p)) at r') q'' prg h' :- t' ;

The actual call to the closure represented in the register r is performed by the code macro 'call-sequence', C.2.4.(1). Let us consider the execution of the first 8 instructions of this code macro, up to the 'call' instruction.

- spare-final 8 m'_p sum(l,n,n',8) nw
  = spare-final 8 (prg, sum(l,n,n'), cz', cn', g', w', q''') sum(l,n,n',8) nw
  = sparc-final 7 (prg, sum(l,n,n',1), cz', cn',
  overlay(map of sp to sum(g' at sp,1), g'), w', q''')
  sum(l,n,n',8) nw
  = spare-final 6 (prg, sum(l,n,n',2), cz', cn',
  overlay(map of sp to sum(g' at sp,1),
  map of global to (q''' at heap) at sum((head of w') at r,1), g'),
  w', q''')
  sum(l,n,n',8) nw
  = spare-final 5 (prg, sum(l,n,n',3), cz', cn',
  overlay(map of sp to sum(g' at sp,1),
  map of global to (q''' at heap) at sum((head of w') at r,1), g'),
  w',
  overlay(map of stack to
  overlay(map of sum(g' at sp,1) at sp to
  (q''' at heap) at sum((head of w') at r,1),

```
            $q'''$ at stack),
          $q'''$))
      sum($l$,$n$,$n'$,8) $nw$
= sparc-final 4 ($prg$, sum($l$,$n$,$n'$,4), $cz'$, $cn'$,
      overlay(map of sp to sum($g'$ at sp,1),
          map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
          $g'$ ),
      $w'$,
      overlay(map of stack to
          overlay(map of sum($g'$ at sp,1) at sp to
              ($q'''$ at heap) at sum((head of $w'$) at $r$,1),
            $q'''$ at stack),
          $q'''$))
      sum($l$,$n$,$n'$,8) $nw$
= spare-final 3 ($prg$, sum($l$,$n$,$n'$,5), $cz'$, $cn'$,
      overlay(map of sp to sum($g'$ at sp,1),
          map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
          map of arg to (head of $w'$) at $r'$,
          $g'$) ,
      $w'$,
      overlay(map of stack to
          overlay(map of sum($g'$ at sp,1) at sp to
              ($q'''$ at heap) at sum((head of $w'$) at $r$, 1),
            $q'''$ at stack),
          $q'''$))
      sum($l$,$n$,$n'$,8) $nw$
= spare-final 2 ($prg$, sum($l$,$n$,$n'$,6), $cz'$, $cn'$,
      overlay(map of sp to sum($g'$ at sp,1),
          map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
          map of arg to (head of $w'$) at $r'$,
          $g'$) ,
      concatenation(list of empty-map, $w'$),
      overlay(map of stack to
          overlay(map of sum($g'$ at sp,1) at sp to
              ($q'''$ at heap) at sum((head of $w'$) at $r$, 1),
            $q'''$ at stack),
          $q'''$))
```

```
    sum($l$,$n$,$n'$,8) $nw$
= spare-final 1 ($prg$, sum($l$,$n$,$n'$,7), $cz'$, $cn'$,
    overlay(map of sp to sum($g'$ at sp,1),
        map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
        map of arg to (head of $w'$) at $r'$,
        $g'$) ,
    concatenation(map of statlink to sum($g'$ at sp,1), $w'$),
    overlay(map of stack to
        overlay(map of sum($g'$ at sp,1) at sp to
            ($q'''$ at heap) at sum((head of $w'$) at $r$, 1),
          $q'''$ at stack),
        $q'''$))
    sum($l$,$n$,$n'$,8) $nw$
= spare-final 0 ($prg$, sum($l$,$n$,$n'$,8), $cz'$, $cn'$,
    overlay(map of sp to sum($g'$ at sp,1),
        map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
        map of arg to (head of $w'$) at $r'$,
        $g'$) ,
    update concatenation(map of statlink to sum($g'$ at sp,1), $w'$),
        (map of $r'$ to (head of $w'$) at $r'$),
    overlay(map of stack to
        overlay(map of sum($g'$ at sp,1) at sp to
            ($q'''$ at heap) at sum((head of $w'$) at $r$, 1),
          $q'''$ at stack),
        $q'''$))
    sum($l$,$n$,$n'$,8) $nw$
= ($prg$, sum($l$,$n$,$n'$,8), $cz'$, $cn'$,
    overlay(map of sp to sum($g'$ at sp,1),
        map of global to ($q'''$ at heap) at ((head of $w'$) at $r$),
        map of arg to (head of $w'$) at $r'$,
        $g'$) ,
    update concatenation(map of statlink to sum($g'$ at sp,1), $w'$),
        (map of $r'$ to (head of $w'$) at $r'$),
    overlay(map of stack to
        overlay(map of sum($g'$ at sp,1) at sp to
            ($q'''$ at heap) at sum((head of $w'$) at $r$, 1),
          $q'''$ at stack),
```

q'''))

In this state, the 'call' instruction will be executed. In the new state, the program counter will be '(q''' at heap) at ((head of w') at r)' and the register window will have a return-address being 'sum(l,n,n',8)'.

It is now straightforward to see that the induction hypothesis point (1) can be applied. The code for the action incorporated in the closure may be placed anywhere in the program *prg*. The program counter may even at some point during execution of the call assume one of the values in '(l_n, l_e, l_f)'. This does *not* cause the execution to yield 'nothing', however, (recalling the definition of 'spare-final' in B.3.1.(1)) because the length of the register window will be strictly greater than nw.

By applying the induction hypothesis point (1) to the action incorporated in the closure, and then executing the remaining 18 instructions of 'call-sequence', it follows that the conclusion of the lemma is satisfied. We will not give the details of this computation, however, but merely note that it is in the remaining 18 instructions where it is crucial that the 'cef' has the correct value after the call.

As the second case of the induction step, consider the 'a-count', 'perform', and 'final' clauses for ⟦ "unfolding" U:Unf ⟧ in C.3.1.(17), C.4.1.(16), and A.3.1.(17).

- (1) u-count U h d false () false () empty-list = ac-state n z_n h_n z_e h_e e
  ⇒ a-count ⟦ "unfolding" U:Unf ⟧ h d = ac-state sum(4,n,18) z_n h_n z_e h_e e .

- (1) u-count U h d false () false () empty-list = ac-state n z_n h_n z_e h_e e ;
  (2) a' = free-register f ;
  (3) l' = sum(l,4) ;
  (4) l'' = sum(l',n) ;
  (5) unf U h a' union(f,set of a) d u_n u_e u_f a' false () a false () a empty-list
  l' l'' sum(l'',6) sum(l'',12) l' = a-state p a_n a_e ;
  (6) either(e is empty-list, u_n is 0) = true
  ⇒ perform ⟦ "unfolding" U:Unf ⟧ h a f d u_n u_e u_f l l_n l_e l_f
  = ac-state overlay(
  putcommit l 0 ,
  map of sum(l,3) to (move a to a') ,

p ,
combine $l'' \; l_n \; l_e \; l_f$)

$a_n \; a_e$.

- final ⟦ "unfolding" U:Unf ⟧ $t \; b \; s \; io$ = unf-final U ⟦ "unfolding" U ⟧ $t \; b$
  $s \; io$.

The code places a representation of 'uncommitted' on top of the stack before entering the unfolding. It also copies the representation of the received data, contained in in register a, to a free register a' and freezes a. We will not give the details of this computation but merely note that by applying the induction hypothesis, point (2), we obtain a 'sparc-state' that satisfies the 'u-post-condition' and not the 'a-post-condition', as required. The subtlety is that after the execution of the loop, the commitment stack contains *two* values whose disjunction is the wanted commitment value. The task of removing the two values and placing their disjunction on top of the commitment stack is performed by the code macro 'combine'.

As the third case of the induction step, consider the 'a-count', 'perform', and 'final' clauses for ⟦ A:Act "then" A':Act ⟧ in C.3.1.(19), C.4.1.(18), and A.3.1.(21).

- (1) a-count $A \; h \; d$ = ac-state $n' \; z'_n \; h'_n \; z'_e \; h'_e$ empty-list ;
  (2) a-count $A' \; h'_n \; d$ = ac-state $n'' \; z''_n \; h''_n \; z''_e \; h''_e \; e$;
  (3) compare-data-types $z'_e \; h'_e \; z''_e \; h''_e$ = h_e:data-type
  ⇒ a-count ⟦ A:Act "then" A':Act ⟧ $h \; d$ = ac-state sum(n',2,n'',l8)
  both(z'_n,z''_n) h''_n either(z'_e,z''_e) $h_e \; e$.
- (1) a-count $A \; h \; d$ = ac-state $n' \; z'_n \; h'_n \; z'_e \; h'_e$ empty-list ;
  (2) a-count $A' \; h'_n \; d$ = ac-state $n'' \; z''_n \; h''_n \; z''_e \; h''_e \; e$;
  (3) l' = sum(l,n') ;
  (4) l'' = sum(l',2,n'') ;
  (5) perform $A \; h \; a \; f \; d$ 0 $u_e \; u_f \; l$ sum(l',2) $l' \; l_f$ = a-state $p'' \; a''_n \; a''_e$;
  (7) either(e is emptylist, u_n is 0) = true
  ⇒ perform ⟦ A:Act "then" A':Act ⟧ $h \; a \; f \; d \; u_n \; u_e \; u_f \; l \; l_n \; l_e \; l_f$
  = a-state overlay(
  p'

map of l' to (move a'_e to a''_e),
map of sum(l',1) to (jump l_e),
p'' ,
combine l'' l_n l_e l_f)
a''_n a''_e

- (1) final A t b s io = completed t' empty-map s' io' c' ;
  (2) final A' t' b s' io' = completed t'' b'' s'' io'' c''
  $\Rightarrow$ final $[\![$ A:Act "then" A':Act $]\!]$ t b s io := completed t'' b'' s'' io'' either(c', c'') .

By applying the induction hypothesis point (1) to the action A we get 'm_p:sparc-state' and 'j:natural' such that

- sparc-final j (p, l, cz, cn, g, w, q) (sum(l',2), l', l_f) $nw = m_p$.
- a-post-condition g w q m_p f 0 u_e u_f e c' = true .
- (1) m-earlier m_p m'_p = true
  $\Rightarrow$ m-abs m'_p z'_n h'_n z'_e h'_e a'_n a'_e empty-list sum(l',2) l' l_f :- (completed t' empty-map s' io' c') .

By using one of the clauses for 'm-abs' namely D.2.(15.5) we come in a situation where we can and use the induction hypothesis point (1) to the action A'. This gives us 'm'_p:sparc-state' 'j':natural' such that, after a simplification, we have

- sparc-final sum(j,j') (p, l, cz, cn, g, w, q) (l'', sum(l'',6), sum(l'',12)) $nw = m'_p$.
- a-post-condition g w q m'_p f u_n u_e u_f e c'' = true .
- (1) m-earlier m_p m'_p = true
  $\Rightarrow$ m-abs m'_p z''_n h''_n z''_e h''_e a''_n a''_e e l'' sum(l'',6) sum(l'',12) :- (completed t'' b'' s'' io'' c'') .

It is now straight-forward to combine the information from the two applications of the induction hypothesis to get *almost* the desired conclusion. The only thing left, then, is that we need the representation of the two commitment values c' and c'' on top of the commitment stack to be replaced by their disjunction. This is done by the code macro 'combine'.

As the fourth case of the induction step, consider the 'u-count', 'unf' and 'unf-final' clauses for "unfold" in C3.2.(6), C.4.2.(6), and A.3.2.(13).

- u-count “unfold” $h\ d\ z_n\ h_n\ z_e\ h_e\ e$ = ac-state 2 $z_n\ h_n\ z_e\ h_e\ e$.
- (1) either(e is empty-list, u_n is 0) = true

  $\Rightarrow$ unf “unfold” $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$

  = a-state overlay(

  map l to (move a to a'), map of sum(l,1) to (jump l_u))

  $a_n\ a_e$.
- unf-final “unfold” ⟦ “unfolding” U:Unf ⟧ $t\ b\ s\ io$ = final ⟦ “unfolding” U ⟧ $t\ b\ s\ io$.

Before jumping to the start of the code for the unfolding, the code copies the representation of the received data, contained in the register a, to the dedicated register a' as explained in chapter 3. To prove the conclusion, we need the ‘final’ clause for ⟦ “unfolding” U ⟧ in A.3.1.(17) that is

- unf-final “unfold” ⟦ “unfolding” U:Unf ⟧ $t\ b\ s\ io$ = final ⟦ unfolding” U ⟧ $t\ b\ s\ io$.
- final ⟦ “unfolding” U:Unf ⟧ $t\ b\ s\ io$ = unf-final U ⟦ “unfolding” U ⟧ $t\ b\ s\ io$;

Using this clause and the ‘unf-final’ clause for “unfold” we can bring us in a situation where we can apply the induction hypothesis, point (2). This immediately yields the desired result. □

E.5 Code Well-behavedness

needs: **Auxiliary Notation ,**
Compiler Consistency ,
Correctness of Analysis .

Lemma: **(Read-only Code)**

n , nw : natural ;
m_p : spare-state ;
lt : linenumber*

$\Rightarrow$

(1) (1) sparc-final n m_p lt $=$ m'_p:sparc-state t
$\Rightarrow$ (program of m_p) is (program of m'_p) $=$ true .

Proof: Consider the definition of 'spare-final _ _ _ _' in B.3.1.(1) that is

```
•  spare-final $n$:atural $m_p$:sparc-state $lt$:linenumber$^*$ $nw$:natural =
       if both($n$ is 0, finished $m_p$ $lt$ $nw$)
       then $m_p$
       else if not any($n$ is 0, finished $m_p$ $lt$ $nw$, out-of-bound $m_p$ )
           then spare-final predecessor($n$) step($m_p$) $lt$ $nw$
           else nothing .
```

The proof of the lemma is by induction on the length of inference of (1.1).

In the base case, we consider inferences of length one, so the rule just given has only been applied once. Since 'nothing' is not an individual, we have '$m'_p = m_p$', from which the conclusion is immediate.

In the induction step, we have the recursive application 'spare-final predecessor(n) step(q) lt nw'. It is a condition that n is non-zero, so its predecessor is an individual. Furthermore, 'step(m_p)' has t he same program component as m_p (this is easily checked), so by applying the induction hypothesis we get the conclusion. □

Lemma: (Code Well-behavedness)

g : globals ;
w : windows ;
j , n , n' , n'' , nw : natural ;
q , q' , q'' memory ;
p , p' , p'' : program ;
h , h' , h_n , h'_n , h_e , h'_e : data-type ;
t : data ;
d : symbol-table ;
b : bindings ;
e , e' : block ;
m_p , m'_p , m''_p : sparc-state ;
m_a : state ;
cz : was-zero ;
cn : was-negative ;

z_n , z'_n , z_e , z'_e : truth-value ;
r , r' , r'' , a , a' , a_n , a'_n , a_e , a'_e : general-register ;
l , l' linenumber ;
l_n :linenumber-complete ;
l_e : linenumber-escape ;
l_f :linenumber-fail ;
l_u : linenumber-unfold ;
S , S' : type ;
v , v' :datum ;
io : input-output ;
s : storage ;
f , f' : frozen ;
u_n , u_e , u_f : cleanup ;

$\Rightarrow$

(1) (1) a-count $A\ h\ d$ = ac-state $n'\ z_n\ h_n\ z_e\ h_e\ e$;
(2) perform A:Act $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f$ = a-state $p'\ a_n\ a_e$;
(3) t-abs ((head of w) at a) $q'\ p\ h$:- t ;
(4) b-abs ((head of w) at staticlink) $q'\ p\ d$:- b ;
(5) store-abs (g at firstfree) (q at store) :- s ;
(6) io:input-output = (il:[integer] list, ol:[integer] list) ;
(7) i-abs ((q at input) at 0) (q at input) = il ;
(8) o-abs ((q at output) at 0) (q at output) = ol ;
(9) pre-condition $p'\ p$ sum(l,n') (l_n, l_e, l_f) $g\ w\ q\ q'\ nw\ d$ = true ;
(10) sum(l,n') leq l' = true ;
(11) spare-final n (p, l, cz, cn, g,w,q) $l'\ nw = m''_p$
$\Rightarrow$ $\exists\ m_p$:sparc-state $\exists\ j$:natural $\exists\ m_a$:state
(12) spare-final j (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) $nw = m_p$;
(13) j leq n = true ;
(14) a-post-condition $g\ w\ q\ m_p\ f\ u_n\ u_e\ u_f\ e$ (commitment of m_a) = true ;
(10) (1) m-earlier $m_p\ m'_p$ = true
$\Rightarrow$ m-abs $m'_p\ z_n\ h_n\ z_e\ h_e\ a_n\ a_e\ e\ l_n\ l_e\ l_f$:- m_a ;
(2) (1) u-count $U\ h\ d\ z_n\ h_n\ z_e\ h_e\ e$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e'$;
(2) u-count $U'\ h'\ d$ false () false () empty-list = ac-state $n''\ z'_n\ h'_n\ z'_e\ h'_e\ e'$;
(3) unf U:Unf $h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$ = a-state

p' a'_n a'_e ;

(4) unf U':Unf h' a' f' d u_n u_e u_f a' false () a' false () a' empty-list l_u l_n l_e l_f l_u = a-state p'' a'_n a'_e ;

(5) t-abs ((head of w) at a) q' p h :- t ;

(6) b-abs ((head of w) at staticlink) q' p d :- b ;

(7) store-abs (g at firstfree) (q at store) :- s ;

(8) io:input-output = (il:[integer] list, ol:[integer] list) ;

(9) i-abs ((q at input) at 0) (q at input) = il ;

(10) o-abs ((q at output) at 0) (q at output) = ol ;

(11) pre-condition p' p sum(l,n') (l_n, l_e, l_f) g w q q' nw d = true ;

(12) p'' is submap of p = true ;

(13) sum(l_u, n'') leq (l_n, l_e, l_f) = true ;

(14) a' is in f = false ;

(15) sum(l,n') leq l' = true ;

(16) spare-final n (p, l, cz, cn, g, w, q) l' nw = m_p ;

⇒ ∃ m_p:sparc-state ∃ j:natural ∃ m_a:state

(17) spare-final j (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) nw = m_p ;

(18) j leq n = true ;

(19) u-post-condition g w q m_p f h u_e u_f e (commitment of m_a) = true ;

(20) (1) m-earlier m_p m'_p = true

⇒ m-abs m'_p z'_n h'_n z'_e h'_e a'_n a'_e e' l_n l_e l_f :- m_a ;

(3) (1) d-count D h d = (n':natural, S:type) ;

(2) evaluate D:Dependent h a f d l l_f = (p':program, r:general-register) ;

(3) t-abs ((head of w) at a) q' p h :- t ;

(4) b-abs ((head of w) at staticlink) q' p d :- b ;

(5) store-abs (g at firstfree) (q at store) :- s ;

(6) pre-condition p' p sum(l, n') l_f g w q q' nw d = true ;

(7) sum(l, n') leq l' = true ;

(8) sparc-final n (p, l, cz, cn, g, w, q) l' nw = m'_p

⇒ ∃ m_p:sparc-state ∃ j:natural

either ∃ v:datum

(1) spare-final j (p, l, cz, cn, g, w, q) sum(l, n') nw = m_p ;

(2) j leq n = true ;

(3) d-post-condition $g\ w\ q\ m_p\ r'$ = true ;

(4) mq-earlier $m_p\ q''$ = true $\Rightarrow$
v-abs ((head of (windows of m_p)) at r) $q''\ p\ S'$:- v' ;

or (1) sparc-final j (p, l, cz, cn, g, w, q) $l_f\ nw = m_p$;

(2) j leq n = true ;

(3) evaluated $D\ t\ b\ s$ = nothing ;

(4) d-post-condition $g\ w\ q\ m_p\ f$ = true ;

(4) (1) w-count $D\ h\ d$ = wc-state n':natural h':data-type;

(2) with D:Tuple $h\ a\ f\ d\ l\ l_f$ = (p':program, r:general-register) ;

(3) t-abs ((head of w) at a) $q'\ p\ h$:- t ;

(4) b-abs ((head of w) at staticlink) $q'\ p\ d$:- b ;

(5) store-abs (g at firstfree) (q at store) :- s ;

(6) pre-condition $p'\ p$ sum(l, n') $l_f\ g\ w\ q\ q'\ nw\ d$ = true ;

(7) sum(l, n') leq l' = true ;

(8) sparc-final n (p, l, cz, cn, g, w, q) $l'\ nw = m'_p$

$\Rightarrow$ $\exists\ m_p$:sparc-state $\exists\ j$:natural

either $\exists\ t'$:data

(1) spare-final j (p, l, cz, cn, g, w, q) sum(l, n') $nw = m_p$;

(2) j leq n = true ;

(3) d-post-condition $g\ w\ q\ m_p\ f$ = true ;

(4) mq-earlier $m_p\ q''$ = true $\Rightarrow$
t-abs ((head of (windows of m_p)) at r) $q''\ p\ h'$:- t' ;

or (1) sparc-final j (p, l, cz, cn, g, w, q) $l_f\ nw = m_p$;

(2) j leq n = true ;

(3) multi-evaluated $D\ t\ b\ s$ = nothing ;

(4) d-post-condition $g\ w\ q\ m_p\ f$ = true ;

(5) (1) find-count $k\ d$ = (n':natural, S:type, j':natural) ;

(2) find k:token $d\ l = p'$:program ;

(3) b-abs (g at global) $q'\ p\ S$:- b ;

(4) basic-pre-condition $p'\ p$ sum(l,n') $l_f\ g\ w\ q\ q'\ nw$ = true

(5) sum(l,n') leq l' = true ;

```
        (6)  sparc-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l'$ $nw = m'_p$
        ⇒    $\exists$ $m_p$:sparc-state $\exists$ $j$:natural $\exists$ $v$:datum
        (7)  sparc-final $j$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l$, $n'$) $nw = m_p$ ;
        (8)  f-post-condition $g$ $w$ $q$ $m_p$ = true ;
        (9)  (1)  mq-earlier $m_p$ $q''$ = true
        ⇒    v-abs sum((globals of $m_p$) at globals, $j'$) $q''$ $p$ $S$ :- $v$ ;
(6)     (1)  unary-count $O$ $S$ = ($n'$:natural, $S'$:type) ;
        (2)  unary-code $O$:Unary $r$ $r'$ $l$ $l_f$ = $p'$:program ;
        (3)  v-abs ((head of $w$) at $r$) $q'$ $p$ $S$ :- $v$ ;
        (4)  pre-condition $p'$ $p$ sum($l$, $n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
        (5)  sum($l$, $n'$) leq $l'$ = true ;
        (6)  sparc-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l'$ $nw = m'_p$
        ⇒    $\exists$ $m_p$:sparc-state $\exists$ $j$:natural
        either $\exists$ $v'$:datum
             (1)  sparc-final $j$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l$, $n'$) $nw = m_p$ ;
             (2)  $j$ leq $n$ = true ;
             (3)  op-post-condition $g$ $w$ $q$ $m_p$ $r'$ = true ;
             (4)  mq-earlier $m_p$ $q''$ = true ⇒
                      v-abs ((head of (windows of $m_p$)) at $r$) $q''$ $p$ $S$ :- $v'$ ;
        or   (1)  sparc-final $j$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l_f$ $nw = m_p$ ;
             (2)  $j$ leq $n$ = true ;
             (3)  unary-operation $O$ $v$ = nothing ;
             (4)  op-post-condition $g$ $w$ $q$ $m_p$ $r'$ = true ;
(7)     (1)  binary-count $O$ $S$ $S'$ = ($n'$:natural, $S''$:type) ;
        (2)  binary-code $O$:Binary $r$ $r'$ $r''$ $l$ $l_f$ = $p'$:program ;
        (3)  v-abs ((head of $w$) at $r$) $q'$ $p$ $S$ :- $v$ ;
        (4)  v-abs ((head of $w$) at $r'$) $q'$ $p$ $S'$ :- $v'$ ;
        (5)  precondition $p'$ $p$ sum($l$,$n'$) $l_f$ $g$ $w$ $q$ $q'$ $nw$ $d$ = true ;
        (6)  sum($l$, $n'$) leq $l'$ = true ;
        (7)  sparc-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l'$ $nw = m'_p$
        ⇒    $\exists$ $m_p$:sparc-state $\exists$ $j$:natural
        either $\exists$ $v''$:datum
```

(1) sparc-final j (p, l, cz, cn, g, w, q) sum(l, n') $nw = m_p$;

(2) j leq n = true ;

(3) op-post-condition $g\ w\ q\ m_p\ r''$ = true ;

(4) mq-earlier $m_p\ q''$ = true $\Rightarrow$
  v-abs ((head of (windows of m_p)) at r) $q''\ p\ S''$:- v'' ;

or (1) sparc-final j (p, l, cz, cn, g, w, q) $l_f\ nw = m_p$;

(2) j leq n = true ;

(3) binary-operation $O\ v\ v'$ = nothing ;

(4) op-post-condition $g\ w\ q\ m_p\ r''$ = true .

Proof: The proof has the same structure as the proof of completeness. Furthermore, most of the details are similar, so we will demonstrate only a single case of the induction step.

Consider the 'a-count' and 'perform' clauses for ⟦ A:Act "then" A':Act ⟧ in C.3.1(19) and C.4.1.(18).

- (1) a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e$ empty-list ;
  (2) a-count $A'\ h'_n\ d$ = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e$;
  (3) compare-data-types $z'_e\ h'_e\ z''_e\ h''_e = h_e$:data-type
  $\Rightarrow$ a-count ⟦ A:Act "then" A':Act] $h\ d$ = ac-state sum(n',2,n'',18)
    both(z'_n,z''_n) h''_n either(z'_e,z''_e) $h_e\ e$.

- (1) a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e$ empty-list ;
  (2) a-count $A'\ h'_n\ d$ = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e$;
  (3) l' = sum(l, n') ;
  (4) l'' = sum(l', 2, n'') ;
  (5) perform $A\ h\ a\ f\ d$ 0 $u_e\ u_f\ l$ sum(l, 2) $l'\ l_f$ = a-state $p'\ a'_n\ a'_e$;
  (6) perform $A'\ h'_n\ a'_n\ f\ d\ u_n\ u_e\ u_f$ sum(l', 2) l'' sum(l'', 6) sum(l'', 12) =
    a-state $p''\ a''_n\ a''_e$;
  (7) either(e is empty-list, u_n is 0) = true
  $\Rightarrow$ perform ⟦ A:Act "then" A':Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ {}_n\ l_e\ l_f$ =
    a-state overlay(p',
    map of l' to (move a'_e to a''_e),
    map of sum(l',1) to (jump l_e),
    p'' ,

$\mathsf{combine}\ l''\ l_n\ l_e\ l_f\ )$

$a''_n\ a''_e$.

Note that in the statement of the lemma, l' is a linenumber past "the end" of the code for $[\![\ A\ \text{“}\mathsf{then}\text{”}\ A'\ ]\!]$, and it is not one of those that will make spare-final yield a result.

By applying the induction hypothesis point (1) to the action A we get 'm_p:sparc-state', 'j:natural', and 'm_a:state' such that

- sparc-final j $(p,\ l,\ cz,\ cn,\ g,\ w,\ q)$ (sum(l',2), l', l_f) $nw = m_p$.

- j leq n = true .

- a-post-condition $g\ w\ q\ m_p\ f$ 0 $u_e\ u_f\ e\ c'$ = true.

  (1) m-earlier $m_p\ m'_p$ = true
  $\Rightarrow$ m-abs $m'_p\ z'_n\ h'_n\ z'_e\ h'_e\ a'_n\ a'_e$ empty-list sum(l' ,2) $l'\ l_f$:- m_a .

Note that n is the number of steps that has been used to pass "the end" of the code for A.

Now there are three cases, one for each linenumber in the tuple '(sum(l', 2), l', l_f)'. We will demonstrate the treatment of only the first of them, the treatment of the others are simpler in that they don't require the use of the induction hypothesis.

In this first case, the code has, after the execution of j steps, reached the line with the number '(sum(l', 2)'.

By using one of the clauses for 'm-abs' namely D.2.(15.5) we *almost* come in a situation where we can apply the induction hypothesis point (1) to the action A'. It is still necessary, though, to demonstrate that there exits a natural number j'' so that

- sparc-final $j''\ m_p\ l'\ nw = m''_p$

Here, l' and m''_p are the same as in the lemma's assumptions. In other words, we must find j'' so that continuing the execution after the first j step will lead to the same state as after n steps. Since 'j leq n = true' we can choose

- j'' = difference(n, j) .

By applying the induction hypothesis point (1) to the action A' we get 'm'_p:sparc-state', 'j':natural', and m'_a:state such that, after a simplification, we have

- sparc-final sum(j,j') (p, l, cz, cn, g, w, q) (l'', sum(l', 6), sum(l'', 12)) $nw = m'_p$.
- j leq n = true .
- j' leq difference(n, j) = true .
- a-post-condition $g\ w\ q\ m'_p\ f\ u_n\ u_e\ u_f\ e\ c''$ = true .
- (1) m-earlier $m_p\ m'_p$ = true
- $\Rightarrow$ m-abs $m'_p\ z''_n\ h''_n\ z''_e\ h''_e\ a''_n\ a''_e\ e\ l''$ sum(l'', 6) sum(l'', 12) :- m'_a .

It is now straight-forward to combine the information from the two applications of the induction hypothesis to get *almost* the desired conclusion. Only two things are left, then. Firstly, we need the representation of the two commitment values c' and c'' on top of the commitment stack to be replaced by their disjunction. This is done by the code macro 'combine'.

Secondly, we need to show that

- sum(j, j') leq n = true .

This follows immediately from 'j' leq difference(n, j) = true'. □

E.6 Soundness

needs: **Auxiliary Notation ,**
Compiler Consistency ,
Correctness of Analysis .
Code Well-behavedness .

Lemma: **(Soundness)**

g : globals ;
w : windows ;
n , n' , n'' , nw : natural ;
q , q' , q'' : memory ;

p , p' , p'' : program ;
h , h' , h_n , h'_n , h_e , h'_e : data-type ;
t : data ;
d : symbol-table ;
b : bindings ;
e , e' : block ;
m_p , m'_p : sparc-state ;
m_a : state ;
cz : was-zero ;
cn : was-negative ;
z_n , z'_n , z_e , z'_e : truth-value ;
r , r' , r'' , a , a' , a_n , a'_n , a_e , a'_e : general-register ;
l : linenumber ;
l_n : linenumber-complete ;
l_e : linenumber-escape ;
l_f : linenumber-fail ;
l_u : linenumber-unfold ;
S , S' : type ;
v , v' : datum ;
io : input-output ;
s : storage ;
f , f' : frozen ;
u_n , u_e , u_f : cleanup ;
$\Rightarrow$

(1) (1) a-count $A\ h\ d =$ ac-state $n'\ z_n\ h_n\ z_e\ h_e\ e$;
(2) perform $A\ h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ l_n\ l_e\ l_f =$ a-state $p'\ a_n\ a_e$;
(3) t-abs ((head of w) at a) $q'\ p\ h$:- t ;
(4) b-abs ((head of w) at staticlink) $q'\ p\ d$:- b ;
(5) store-abs (g at firstfree) (q at store) :- s ;
(6) io:input-output = (il:[integer] list, ol:[integer] list) ;
(7) i-abs ((q at input) at 0) (q at input) = il ;
(8) o-abs ((q at output) at 0) (q at output) = ol ;
(9) precondition $p'\ p$ sum(l, n') (l_n, l_e, l_f) $g\ w\ q\ q'\ nw\ d =$ true ;
(10) spare-final n (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) $nw = m_p$
$\Rightarrow$ $\exists\ m_a$:state

(11) final A:Act $t\ b\ s\ io = m_a$;

(12) a-post-condition $g\ w\ q\ m_p\ f\ u_n\ u_e\ u_f\ e$ (commitment of m_a) = true ;

(13) (1) m-earlier $m_p\ m'_p$ = true

$\Rightarrow$ m-abs $m'_p\ z_n\ h_n\ z_e\ h_e\ a_n\ a_e\ e\ l_n\ l_e\ l_f$:- m_a ;

(2) (1) u-count $U\ h\ d\ z_n\ h_n\ z_e\ h_e\ e$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e\ e'$;

(2) u-count $U'\ h'\ d$ false () false () empty-list = ac-state $n''\ z'_n\ h'_n\ z'_e\ h'_e\ e'$;

(3) unf $U\ h\ a\ f\ d\ u_n\ u_e\ u_f\ a'\ z_n\ h_n\ a_n\ z_e\ h_e\ a_e\ e\ l\ l_n\ l_e\ l_f\ l_u$ = a-state $p'\ a'_n\ a'_e$;

(4) unf $U'\ h'\ a'\ f'\ d\ u_n\ u_e\ u_f\ a'$ false () a' false () a' empty-list $l_u\ l_n\ l_e\ l_f\ l_u$ =
a-state $p''\ a'_n\ a'_e$;

(5) t-abs ((head of w) at a) $q'\ p\ h$:- t ;

(6) b-abs ((head of w) at staticlink) $q'\ p\ d$:- b ;

(7) store-abs (g at firstfree) (q at store) :- s ;

(8) io:input-output = (il[integer] list, ol:[integer] list) ;

(9) i-abs ((q at input) at 0) (q at input) = il ;

(10) o-abs ((q at output) at 0) (q at output) = ol ;

(11) pre-condition $p'\ p$ sum(l, n') (l_n, l_e, l_f) $g\ w\ q\ q'\ nw\ d$ = true ;

(12) p'' is submap of p = true ;

(13) sum(l_u, n'') leq (l_n, l_e, l_f) = true ;

(14) a' is in f = false ;

(15) spare-final n (p, l, cz, cn, g, w, q) (l_n, l_e, l_f) $nw = m_p$

$\Rightarrow$ $\exists\ m_a$:state

(16) unf-final U:unf ⟦ "unfolding" U':unf⟧ $t\ b\ s\ io = m_a$;

(17) u-post-condition $g\ w\ q\ m_p\ f\ u_n\ u_e\ u_f\ e$ (commitment of m_a) = true ;

(18) (1) m-earlier $m_p\ m'_p$ = true

$\Rightarrow$ m-abs $m'_p\ z'_n\ h'_n\ z'_e\ h'_e\ a'_n\ a'_e\ e'\ l_n\ l_e\ l_f$:- m_a ;

(3) (1) d-count $D\ h\ d$ = (n':natural, S:type) ;

(2) evaluate $D\ h\ a\ f\ d\ l\ l_f$ = (p':program, r:general-register) ;

(3) t-abs ((head of w) at a) $q'\ p\ h$:- t ;

(4) b-abs ((head of w) at staticlink) $q'\ p\ d$:- b ;

(5) store-abs (g at firstfree) (q at store) :- s ;

(6) pre-condition $p'\ p$ sum(l, n') $l_f\ g\ w\ q\ q'\ nw\ d$ = true ;

(7) sparc-final n (p, l, cz, cn, g, w, q) sum(l, n') $nw = m_p$

```
     ⇒    $\exists\ v$:datum
     (8)  evaluated $D$:Dependent $t\ b\ s = v$:datum ;
     (9)  d-post-condition $g\ w\ q\ m_p\ f =$ true ;
     (10) (1)  mq-earlier $m_p\ q'' =$ true
          ⇒    v-abs ((head of (windows of $m_p$)) at $r$) $q''\ p\ S$ :- $v$ ;
(4)  (1)  d-count $D\ h\ d = (n'$:natural, $S$:type) ;
     (2)  evaluate $D\ h\ a\ f\ d\ l\ l_f = (p'$:program, $r$:general-register) ;
     (3)  t-abs ((head of $w$) at $a$) $q'\ p\ h$ :- $t$ ;
     (4)  b-abs ((head of $w$) at staticlink) $q'\ p\ d$ :- $b$ ;
     (5)  store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
     (6)  pre-condition $p'\ p$ sum($l$, $n'$) $l_f\ g\ w\ q\ q'\ nw\ d =$ true ;
     (7)  sparc-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) $l_f\ nw = m_p$
     ⇒
     (8)  evaluated $D$:Dependent $t\ b\ s =$ nothing ;
     (9)  d-post-condition $g\ w\ q\ m_p\ f =$ true ;
(5)  (1)  w-count $D\ h\ d =$ wc-state $n'$:natural $h'$:data-type ;
     (2)  with $D\ h\ a\ f\ d\ l\ l_f = (p'$:program, $r$:general-register) ;
     (3)  t-abs ((head of $w$) at $a$) $q'\ p\ h$ :- $t$ ;
     (4)  b-abs ((head of $w$) at staticlink) $q'\ p\ d$ :- $b$ ;
     (5)  store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
     (6)  pre-condition $p'\ p$ sum($l$, $n'$) $l_f\ g\ w\ q\ q'\ nw\ d =$ true ;
     (7)  sparc-final $n$ ($p$, $l$, $cz$, $cn$, $g$, $w$, $q$) sum($l$, $n'$) $nw = m_p$
     ⇒    $\exists\ v$:datum
     (8)  multi-evaluated $D$:Tuple $t\ b\ s = t'$:data ;
     (9)  d-post-condition $g\ w\ q\ m_p\ f =$ true ;
     (10) (1)  mq-earlier $m_p\ q'' =$ true
          ⇒    t-abs ((head of (windows of $m_p$)) at $r$) $q''\ p\ h'$ :- $t'$ ;
(6)  (1)  w-count $D\ h\ d =$ wc-state $n'$:natural $h'$:data-type ;
     (2)  with $D\ h\ a\ f\ d\ l\ l_f = (p'$:program, $r$:general-register) ;
     (3)  t-abs ((head of $w$) at $a$) $q'\ p\ h$ :- $t$ ;
     (4)  b-abs ((head of $w$) at staticlink) $q'\ p\ d$ :- $b$ ;
     (5)  store-abs ($g$ at firstfree) ($q$ at store) :- $s$ ;
```

(6) pre-condition p' p sum(l,n') l_f g w q q' nw d = true ;
(7) sparc-final n (p, l, cz, cn, g, w, q) l_f nw = m_p
$\Rightarrow$ $\exists$ v:datum
(8) multi-evaluated D:Tuple t b s = nothing ;
(9) d-post-condition g w q m_p f = true ;

(7) (1) find-count k d = (n':natural, S:type, j:natural) ;
(2) find k d l = p':program ;
(3) b-abs (g at global) q' p S :- b ;
(4) basic-pre-condition p' p sum(l, n') l_f g w q q' nw = true
(5) sparc-final n (p, l, cz, cn, g, w, q) sum(l, n') nw = m_p
$\Rightarrow$
(6) f-post-condition g w q m_p = true ;
(7) (1) mq-earlier m_p q'' = true
$\Rightarrow$ v-abs sum((globals of m_p) at globals, j) q'' p S :- (b at k:token) ;

(8) (1) unary-count O S = (n':natural, S':type) ;
(2) unary-code O r r' l l_f = p':program ;
(3) v-abs ((head of w) at r) q' p S :- v ;
(4) pre-condition p' p sum(l,n') l_f g w q q' nw d = true ;
(5) sparc-final n (p, l, cz, cn, g, w, q) sum(l,n') nw = m_p
$\Rightarrow$ $\exists$ v:datum
(6) unary-operation O:unary v = v':datum ;
(7) op-post-condition g w q m_p r' = true ;
(8) (1) mq-earlier m_p q'' = true
$\Rightarrow$ v-abs ((head of (windows of m_p)) at r') q'' p S' :- v' ;

(9) (1) unary-count O S = (n':natural, S':type) ;
(2) unary-code O r r' l l_f = p':program ;
(3) v-abs ((head of w) at r) q' p S :- v ;
(4) pre-condition p' p sum(l,n') l_f g w q q' nw d = true ;
(5) sparc-final n (p, l, cz, cn, g, w, q) l_f nw = m_p
$\Rightarrow$
(6) unary-operation O:unary v = nothing ;
(7) op-post-condition g w q m_p r' = true ;

(10) (1) binary-count $O\ S\ S' = (n'$:natural, S'':type) ;
(2) binary-code $O\ r\ r'\ r''\ l\ l_f = p'$:program ;
(3) v-abs ((head of w) at r) $q'\ p\ S$:- v ;
(4) v-abs ((head of w) at r') $q'\ p\ S'$:- v' ;
(5) pre-condition $p'\ p$ sum(l, n') $l_f\ g\ w\ q\ q'\ nw\ d =$ true ;
(6) sparc-final n (p, l, cz, cn, g, w, q) sum(l, n') $nw = m_p$
$\Rightarrow$ $\exists\ v''$:datum
(7) binary-operation O:Binary $v\ v' = v''$:datum ;
(8) op-post-condition $g\ w\ q\ m_p\ r'' =$ true ;
(9) (1) mq-earlier $m_p\ q'' =$ true
$\Rightarrow$ v-abs ((head of (windows of m_p)) at r'') $q''\ p\ S'$:- v'' ;

(11) (1) binary-count $O\ S\ S' =$ (n':natural, S'':type) ;
(2) binary-code $O\ r\ r'\ r''\ l\ l_f = p'$:program ;
(3) v-abs ((head of w) at r) $q'\ p\ S$:- v ;
(4) v-abs ((head of w) at r') $q'\ p\ S'$:- v' ;
(5) pre-condition $p'\ p$ sum(l, n') $l_f\ g\ w\ q\ q'\ nw\ d =$ true ;
(6) spare-final n (p, l, cz, cn, g, w, q) $l_f\ nw = m_p$
$\Rightarrow$
(7) binary-operation O:Binary $v\ v' =$ nothing ;
(8) op-post-condition $g\ w\ q\ m_p\ r'' =$ true .

Proof: The proof has the same structure as the proof of completeness. Furthermore, most of the details are similar, except for the repeated application of the code well-behavedness lemma, so we will demonstrate only a single case of the induction step.

Consider the 'a-count', 'perform', and 'final' clauses for ⟦ A:Act "then" A':Act ⟧ in C.3.1.(19), C.4.1.(18), and A.3.1.(21).

- (1) a-count $A\ h\ d =$ ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e$ empty-list ;
  (2) a-count $A'\ h'_n\ d =$ ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e$;
  (3) compare-data-types $z'_e\ h'_e\ z''_e\ h''_e = h_e$:data-type
  $\Rightarrow$ a-count ⟦ A:Act "then" A':Act] $h\ d =$ ac-state sum(n', 2,n'',18)
  both(z'_n, z''_n) h''_n either(z'_e, z''_e) $h_e\ e$.

- (1) a-count $A\ h\ d$ = ac-state $n'\ z'_n\ h'_n\ z'_e\ h'_e$ empty-list ;
  (2) a-count $A'\ h'_n\ d$ = ac-state $n''\ z''_n\ h''_n\ z''_e\ h''_e\ e$;
  (3) l' = sum(l, n') ;
  (4) l'' = sum(l', 2, n'') ;
  (5) perform $A\ h\ a\ f\ d$ 0 $u_e\ u_f\ l$ sum(l, 2) $l'\ l_f$ = a-state $p'\ a'_n\ a'_e$;
  (6) perform $A'\ h'_n\ a'_n\ f\ d\ u_n\ u_e\ u_f$ sum(l', 2) l'' sum(l'', 6) sum(l'', 12) =
        a-state $p''\ a''_n\ a''_e$;
  (7) either(e is empty-list, u_n is 0) = true
  ⇒ perform ⟦ A:Act "then" A':Act ⟧ $h\ a\ f\ d\ u_n\ u_e\ u_f\ l\ _n\ l_e\ l_f$ = a-state
        overlay(p',
        map of l' to (move a'_e to a''_e),
        map of sum(l',1) to (jump l_e),
        p'' ,
        combine $l''\ l_n\ l_e\ l_f$)

        $a''_n\ a''_e$.

- (1) final $A\ t\ b\ s\ io$ = completed t' empty-map $s'\ io'\ c'$;
  (2) final $A'\ t'\ b\ s'\ io'$ = completed $t''\ b''\ s''\ io''\ c''$
  ⇒ final ⟦ A:Act "then" A':Act ⟧ $t\ b\ s\ io$:= completed $t''\ b''\ s''\ io''$
        either(c', c'') .

By applying the code well-behavedness lemma point (1) to the action A, we get 'm_p:sparc-state' and 'j:natural' such that

- sparc-final j (p, l, cz, cn, g, w, q) (sum(l', 2), l', l_f) $nw = m''_p$.
- j leq n = true .

By using this and applying the induction hypothesis point (1) we further get 'm_a:state' such that

- final $A\ t\ b\ s\ io = m_a$.
- a-post-condition $g\ w\ q\ m''_p\ f$ 0 $u_e\ u_f\ e\ c'$ = true .
- (1) m-earlier $m''_p\ m'_p$ = true
  ⇒ m-abs $m'_p\ z'_n\ h'_n\ z'_e\ h'_e\ a'_n\ a'_e$ empty-list sum(l', 2) $l'\ l_f$:- m_a .

There are now three cases, depending on whether the performance of A has completed, escaped, or failed. We will demonstrate the treatment of only the first of them, the treatment of the others are simpler in that they don't require the use of the induction hypothesis.

In this first case, the code has, after the execution of j steps, reached the line with the number '$\mathsf{sum}(l', 2)$'. From the definition of 'm-abs' in D.2.(15) we get that '$m_a = \mathsf{completed}\ t'\ \mathsf{empty\text{-}map}\ s'\ io'\ c'$'. We also *almost* come in a situation where we can apply code well-behavedness lemma and the induction hypothesis point (1) to the action A'.

To be allowed to apply the code well-behavedness lemma to A', it sufficient to show that there exits a natural number j'' so that

- $\mathsf{spare\text{-}final}\ j''\ m''_p\ (l_n, l_e, l_f)\ nw = m_p$

In other words, we must find j'' so that continuing the execution after the first j step will lead to the same state as after n steps. Since '$j\ \mathsf{leq}\ n = \mathsf{true}$' we can choose

- $j'' = \mathsf{difference}(n, j)$.

By applying the code well-behavedness lemma to the action A' we get 'm'''_p:sparc-state' and 'j':natural' such that, after a simplification, we have

- $\mathsf{sparc\text{-}final}\ \mathsf{sum}(j, j')\ (p, l, cz, cn, g, w, q)\ (l'', \mathsf{sum}(l'', 6), \mathsf{sum}(l'', 12))$
  $nw = m'''_p$.
- $j'\ \mathsf{leq}\ j'' = \mathsf{true}$.
- $\mathsf{a\text{-}post\text{-}condition}\ g\ w\ q\ m'''_p\ f\ u_n\ u_e\ u_f\ e\ c'' = \mathsf{true}$.
- (1) $\mathsf{m\text{-}earlier}\ m'''_p\ m'_p = \mathsf{true}$
  $\Rightarrow$ $\mathsf{m\text{-}abs}\ m'_p\ z''_n\ h''_n\ z''_e\ h''_e\ a''_n\ a''_e\ e\ l''\ \mathsf{sum}(l'', 6)\ \mathsf{sum}(l'', 12)$:- m'_a .

By using this and applying the induction hypothesis point (1) we further get 'm'_a:state' such that, after a simplification, we have

- $\mathsf{final}\ A'\ t'\ b\ s'\ io' = m'_a$.
- $\mathsf{a\text{-}post\text{-}condition}\ g\ w\ q\ m'_p\ f\ u_n\ u_e\ u_f\ e\ c'' = \mathsf{true}$.
- (1) $\mathsf{m\text{-}earlier}\ m_p\ m'_p = \mathsf{true}$
  $\Rightarrow$ $\mathsf{m\text{-}abs}\ m'_p\ z''_n\ h''_n\ z''_e\ h''_e\ a''_n\ a''_e\ e\ l''\ \mathsf{sum}(l'', 6)\ \mathsf{sum}(l'', 12)$:- m'_a .

There are now three cases, depending on whether the performance of A has completed, escaped, or failed. We will demonstrate the treatment of only the first of them, the treatment of the others are similar. In this first case, the code has, after the execution of j steps, reached the line with the number l''. From the definition of '$\mathsf{m\text{-}abs}$' in D.2(15) we get that '$m'_a = \mathsf{completed}\ t''\ b''\ s''\ io''\ c''$'.

In the line l'' we find start of the code macro '$\mathsf{combine}$'. Executing six further steps yields a state where the representation of the two commitment values c' and c'' on top of the commitment stack has been replaced by their disjunction. It is then straight-forward to see, using the above clause for 'final' applied to $[\![\ A\ \text{“}\mathsf{then}\text{”}\ A'\ ]\!]$ that the conclusion follows. □

Appendix F

Main Theorem

needs: **Data Notation ,**
A Compilable Subset of Action Notation ,
A Pseudo SPARC Machine Language ,
Actions to SPARC Compiler ,
Abstraction of Semantic Entities ,
Lemmas.

introduces: result , result _ _ _ _ _ _ _ ,
run _ _ , sparc-run _ _ _ , compile _ , abstract _ _ _ _ _ _ _ .

- result_ _ _ _ _ _ _ :: program, truth-value, data-type, truth-value,
  data-type, general-register, general-register → result (*total*) .

- run _ _ :: Act, [integer] list → state .

- spare-run _ _ _ :: program, natural, space → spare-state .

- compile _ :: Act → result .

- abstract _ _ _ _ _ _ _ :: spare-state, truth-value, data-type, truth-value, data-type,
  general-register, general-register → state .

  z_n , z_e : truth-value ;

h_n , h_e : data-type ;
a_n , a_e : general-register ;
p : program ;
n : natural ;
m_p : spare-state ;
m_a : state ;
se : space ;
il : [integer] list

$\Rightarrow$

(1) run A:Act il = final A () empty-map empty-storage (il empty-list) .

(2) spare-run p n se = spare-final n (p, 0, false, false, overlay(
        map of firstfree to 0, map of sp to 0, map of hp to 2, map of cp to 0),
        list of overlay(map of staticlink to 0, map of (reg 0) to 0),
        overlay(map of stack to empty-map,
                map of store to empty-map,
                map of heap to (map of 1 to -1),
                map of commits to empty-map,
                map of input to se,
                map of output to (map of 0 to 0)))
                (count of elements of mapped-set of p) 1 .

(3) (1) a-count A () (list of empty-list) = ac-state n z_n h_n z_e h_e empty-list ;
    (2) perform A () (reg 0) empty-set (list of empty-list) 0 0 0 0 n n n =
                a-state p a_n a_e .
    $\Rightarrow$ compile A:Act = result p z_n h_n z_e h_e a_n a_e .

(4) (1) n = count of elements of mapped-set of program of m_p ;
    (2) m-abs m_p z_n h_n z_e h_e a_n a_e empty-list n n n :- m_a
    $\Rightarrow$ abstract m_p z_n h_n z_e h_e a_n a_e :- m_a

Theorem:

p : program ;
z_n , z_e : truth-value ;
h_n , h_e : data-type ;
a_n , a_e : general-register ;
se : page

$\Rightarrow$

(1) compile A :Act = result p z_n h_n z_e h_e a_n a_n ;

(2) i-abs (se at 0) $se = il$:[integer] list

```
⇒ (1)   run $A$ $il = m_a$:state ⇒
          ( ∃ $m_p$:sparc-state ∃ $n$:natural .
          spare-run $p$ $n$ $se = m_p$ ;
          abstract $m_p$ $z_n$ $h_n$ $z_e$ $h_e$ $a_n$ $a_e$ :- $m_a$

  (2)   spare-run $p$ $n$:natural $se = m_p$:sparc-state ⇒
          ( ∃ $m_a$:state .
          run $A$ $il = m_a$ ;
          abstract $m_p$ $z_n$ $h_n$ $z_e$ $h_e$ $a_n$ $a_e$ :- $m_a$ ) .
```

Proof: Immediate from the previous lemmas on soundness and completeness, using the compiler consistency lemma for Act and the read-only code lemma. □

Appendix G

HypoPL Action Semantics

G.1 Abstract Syntax

grammar:

(1)	Program	=	⟦ “program” Identifier Block ⟧ .
(2)	Declaration	=	⟦ “int” Identifier ⟧ |
			⟦ “bool” Identifier ⟧ |
			⟦ “const” Identifier “=” Integer ⟧ |
			⟦ “array” Identifier “[” Integer “]” ⟧ |
			⟦ “procedure” Identifier “(” Identifier “)” Block ⟧ |
			⟦ Declaration “;” Declaration ⟧ .
(3)	Block	=	⟦ Declaration “begin” Statement “end” ⟧ |
			⟦ “begin” Statement “end” ⟧ .
(4)	Statement	=	⟦ Expression “:=” Expression ⟧ |
			⟦ “write” Expression ⟧ |
			⟦ “read” Expression ⟧ |
			⟦ “if” Expression “then” Statement “else” Statement “endif” ⟧ |
			⟦ “while” Expression “do” Statement “endwhile” ⟧ |
			⟦ Identifier “(” Expression “)” ⟧ |
			⟦ Statement “;” Statement ⟧ | “skip” .
(5)	Expression	=	“true” | “false” | Integer | Identifier |
			⟦ Identifier “[” Expression “]” ⟧ |
			⟦ Expression Operation Expression ⟧ |

⟦ “not” Expression ⟧ .

(6) Operation = “+” | “−” | “<” | “=” | “and” .

(7) Integer = natural | ⟦ “-” natural ⟧ .

(8) Identifier = token .

G.2 Semantic Entities

G.2.1 Items

introduces: item

(1) truth-value | integer .

G.2.2 Coercion

introduces: coercively _ .

• coercively _ :: act → act .

(1) coercively A:act =

```
| A
then
| give the given item #1 or
| give the item stored in the given cell #1 .
```

G.3 Semantic Functions

introduces: run _ , establish _, activate _, execute _, evaluate _, operation-result _, integer-value _ id _ .

G.3.1 Programs

• run _ :: Program → act .

(1) run ⟦“program” I: identifier B: block ⟧ = activate B .

G.3.2 Declarations

```
•    establish _ :: Declaration → act .
(1)  establish ⟦"int" $I$: identifier ⟧ = allocate integer cell then bind id $I$ to it .
(2)  establish ⟦"bool" $I$: identifier ⟧ = allocate truth-value cell then bind id $I$ to it .
(3)  establish ⟦"const" $I$: identifier "=" $j$: integer ⟧ = bind id $I$ to integer-value $j$ .
(4)  establish ⟦ "array" $I$: Identifier "[" $j$: integer "]" ⟧ =
         | give empty-list & [integer cell] list and then
         | give sum(integer-value $j$, 1)
         then
         | unfolding
         | | | check the given integer #2 is 0 and then
         | | | give the given list #1
         | | or
         | | | | regive and then allocate integer cell
         | | | then
         | | | | | give concatenation(list of the given integer cell #3, the given list #1)
         | | | | and then give difference(the given integer #2, 1)
         | | then
         | | | unfold
         then
         | bind id $I$ to the given list #1 .
(5)  establish ⟦ "procedure" $I_1$: Identifier "(" $I_2$:Identifier ")" $I_3$:Block ⟧ =
         bind id $I_1$ to
         closure abstraction of
         | | furthermore
         | | | | give the given integer #1 and then
         | | | | allocate integer cell
         | | | then
         | | | | store the given integer #1 in the given cell #2 and then
         | | | | bind id $I_2$ to the given cell #2
         | thence activate $B$
         & [perhaps using integer] act .

(6)  establish ⟦ $D_1$:Declaration ";" $D_2$:Declaration ⟧ = establish $D_1$ before establish $D_2$ .
```

G.3.3 Blocks

```
•    activate _ :: Block → act .
(1)  activate ⟦ D: Declaration "begin" S: Statement "end" ⟧ =
        | furthermore establish D
        hence execute S .
(2)  activate ⟦"begin" S: Statement "end" ⟧ = execute S .
```

G.3.4 Statements

```
•    execute _ :: Statement → act .
(1)  execute ⟦ E1: Expression "=:" E2: Expression ⟧ =
        | evaluate E1 and then
        | coercively evaluate E2
        then store the given item #2 in the given cell #1 .
(2)  execute ⟦ "write" E:Expression ⟧ =
          coercively evaluate E then batch-send it .
(3)  execute ⟦ "read" E:Expression ⟧ =
        | batch-receive an integer and then evaluate E
        then store the given integer #1 in the given integer cell #2 .
(4)  execute ⟦ "if" E: Expression "then" S1:Statement "else" S2: Statement "endif" ⟧ =
          | coercively evaluate E
          then
          | | check it then execute S1
          | or
          | | check not it then execute S2
(5)  execute ⟦ "while" E:Expression "do" S:Statement "endwhile" ⟧ =
          unfolding
          | | coercively evaluate E
          | then
          | | | check it then execute S then unfold
          | | or check not it .
(6)  execute ⟦ I: Identifier "(" E: Expression ")" ⟧ =
    | give the abstraction bound to id $I$ and then
    | coercively evaluate $E$
    then enact application the given abstraction #1 to the given integer #2 .
```

(7) execute ⟦ S_1:Statement ";" S_2:Statement ⟧ = execute S_1 and then execute S_2 .

(8) execute "skip" = complete .

G.3.5 Expressions

• evaluate _ :: Expression → act .

(1) evaluate "true" = give true .

(2) evaluate "false" = give false

(3) evaluate i: Integer = give integer-value i

(4) evaluate I: Identifier = give the datum bound to id I .

(5) evaluate ⟦I: Identifier "[" E:Expression "]" ⟧ =

```
    | give the list bound to id $I$ and then
    | | coercively evaluate $E$ then give sum(it, 1)
    then give component# (the given integer #2) items (the given list #1) .
```

(6) evaluate ⟦ E_1: Expression O: Operation E_2: Expression ⟧ =

```
    | coercively evaluate $E_1$ and then
    | coercively evaluate $E_2$
    then give operation-result $O$ .
```

(7) evaluate ⟦ "not" E: Expression ⟧ =

```
      coercively evaluate $E$ then
      give not it .
```

G.3.6 Operations

• operation-result _ :: Operation → dependent datum .

(1) operation-result "+" = sum(the given integer #1, the given integer #2) .

(2) operation-result "−" = difference(the given integer #1, the given integer #2) .

(3) operation-result "<" = (the given integer #1) is less than (the given integer #2) .

(4) operation-result "=" = (the given item 1 cell #1) is (the given item [cell #2) .

(5) operation-result "and" = both(the given truth-value #1, the given truth-value #2) .

G.3.7 Integers

- integer-value _ :: Integer $\rightarrow$ Integer .

(1) integer-value n: natural $=$ n .

(2) integer-value $[\![$ "$-$" n:natural $]\!]$ $=$ negation n .

G.3.8 Identifiers

- id _ :: Identifier $\rightarrow$ token .

(1) id k: token $=$ k .

Appendix H

The HypoPL Bubble-sort Program

This appendix presents the HypoPL bubble-sort program which was used in chapter 3. It also presents excerpts of the action and assembly code generated from this program.

```
program bubblesort
/*
 * This HypoPL program performs a sorting exercise for timing purposes.
 *
 * input:   an integer n, n ≤ 1000
 * action:  sorts 0..n-1 into descending order using bubble-sort
 * output: original array: 0..n−1
 *            marker : −999
 *            sorted array : n−1..0
 */

const maxsize = 1000 ;
array num [1000] ;      /* The array to sort */

procedure sort (numElts)
/*
 * This procedure uses a bubble sort to arrange the 0..numElts−1
 * elements of the "num" array
 */
```

```
   int last ;
   int current ;
   int temp ;

   begin
        last := numElts −1 ;
        while 0 < last do
             current : = 0 ;

             while  current < last do
                  if  num [current] < num [current + 1] then
                       temp : = num [current] ;
                       num [current] : = num [current + 1] ;
                       num [current + 1] : = temp
                  else
                       skip
                  endif ;
                  current : = current + 1
             endwhile ;

             last : = last −1
        endwhile
   end ; /* sort */

procedure printNums (size)
/*
 * Print out the "num" array, 0..size −1
 */
   int i

   begin
        i := 0 ;
        while i < size do
             write num [i] ;
             i := i + 1
        endwhile
   end ; /* printNums */
```

```
/*
 * Main program
 */


inti ;
intnumSort     /* The number of integers to sort */

begin
   read  numSort ;

   if  max_size < numsort
   then  skip
   else
       i := 0 ;

       while i < numSort do
           num [i] := i ;
           i := i + 1
       endwhile ;

       printNums (numSort) ;
       write − 999 ;
       sort (numSort) ;
       print Nums (numSort)
   endif
end
```

The following is the action generated for the printNums procedure.

```
establish ⟦ "procedure" "printNums" "(" "size" ")" . . . ⟧ =
 bind "printnums" to closure abstraction of
 | furthermore
 | | | give the given integer #1 and then allocate integer cell
 | | then
 | | | | store the given integer #1 in the given cell #2
 | | | and then
 | | | | bind "size" to the given cell #2
 | thence
 | | furthermore
 | | | allocate integer cell then bind "i" to it
 | | hence
 | | | | | | give the datum bound to "i"
 | | | | | and then
 | | | | | | | give 0
 | | | | | | then
 | | | | | | | give the given (truth-value | integer) #1 or
 | | | | | | | give the (truth-value | integer) stored in the given cell #1
 | | | | then
 | | | | | store the given (truth-value | integer) #2 in the given cell #1
 | | | and then
 | | | | unfolding
 | | | | | | | | give the datum bound to "i"
 | | | | | | | then
 | | | | | | | | give the given (truth-value 1 integer) #1 or
 | | | | | | | | give the (truth-value | integer) stored in the given cell #1
 | | | | | | and then
 | | | | | | | | give the datum bound to "size"
 | | | | | | | then
 | | | | | | | | give the given (truth-value | integer) #1 or
 | | | | | | | | give the (truth-value | integer) stored in the given cell #1
 | | | | | then
 | | | | | | give the given integer #1 is less than the given integer #2
 | | | | | then
 | | | | | | give the given (truth-value | integer) #1 or
 | | | | | | give the (truth-value | integer) stored in the given cell #1
```

```
│ │ │ │ │ then
│ │ │ │ │ │ │ │ check it
│ │ │ │ │ │ │ then
│ │ │ │ │ │ │ │ execute ⟦ "write" "num" "[" "i" "]" ";" "i" ":=" "i" "+" 1 ⟧
│ │ │ │ │ │ │ then
│ │ │ │ │ │ │ │ unfold
│ │ │ │ │ │ or
│ │ │ │ │ │ │ check not it
& [ perhaps using integer ] act
```

```
execute ⟦ “write” “num” “[” “i” “]” “;” “i” “:=” “i” “+” 1 ⟧
  | | | give the list bound to num
  | | and then
  | | | | give the datum bound to “i”
  | | | then
  | | | | give the given (truth-value | integer) #1 or
  | | | | give the (truth-value | integer) stored in the given cell #1
  | | | then
  | | | | give sum(it,1)
  | then
  | | give component# (the given integer #2) items (the given list #1)
  | then
  | | give the given (truth-value | integer) #1 or
  | | give the (truth-value | integer) stored in the given cell #1
  | then
  | | batch-send it
  and then
  | | | give the datum bound to “i”
  | | and then
  | | | | | | give the datum bound to “i”
  | | | | | then
  | | | | | | give the given (truth-value | integer) #1 or
  | | | | | | give the (truth-value | integer) stored in the given cell #1
  | | | | and then
  | | | | | | give 1
  | | | | | then
  | | | | | | give the given (truth-value | integer) #1 or
  | | | | | | give the (truth-value | integer) stored in the given cell #1
  | | | then
  | | | | give sum(the given integer #1, the given integer #2)
  | | | then
  | | | | give the given (truth-value | integer) #1 or
  | | | | give the (truth-value | integer) stored in the given cell #1
  | then
  | | store the given (truth-value | integer) #2 in the given cell #1
```

The following is the action generated for the statement “**write**−999”.

```
execute ⟦ “write” ⟦ 999 ⟧ ⟧
 | give negation 999
 then
 | give the given (truth-value | integer) #1 or
 | give the (truth-value | integer) stored in the given cell #1
 then
 | batch-send it
```

The following is the SPARC code generated for the statement “**write**−999”, when compiled as part of the HypoPL bubble-sort program. The Pseudo Sparc instructions appear as comments (in a concrete syntax that differs slightly from the abstract syntax used in the formal specification). Other comments indicate for which part of the action the code is generated.

```
                                  !  (move 999 to (reg 2))              !!  999
    or     %g0, 999, %r26
                                  !  (move 0 to global)                 !!  negation
    or     %g0, 0, %g1
                                  !  (move difference global (reg 2) to (reg 5))
    sub    %g1, %r26, %r23
                                  !  (store (reg 5) in hp 2 heap)       !!  give)
    set    (_DSpace + 10008), %o0
    st     %r23, [%o0 + %g3]
                                  !  (move -4 to global)
    or     %g0, -4, %g1
                                  !  (store global in hp 1 heap)
    set    (_DSpace + 10004), %o0
    st     %g1, [%o0 + %g3]
                                  !  (store hp in hp 3 heap)
    set    (_DSpace + 10012), %o0
    st     %g3, [%o0 + %g3]
                                  !  (move sum hp 8 to (reg 2))
    add    %g3, 8, %r26
                                  !  (move sum hp 16 to hp)
    add    %g3, 16, %g3
                                  !  (move 0 to global)
    or     %g0, 0, %g1
                                  !  (store global in cp 0 commits)
    set    (_DSpace + 2000), %o0
    st     %g1, C[%o0 + %g4]
                                  !  (move sum cp 4 to cp)
    add    %g4, 4, %g4
                                  !  (move difference sp 0 to sp)
    sub    %g2, 0, %g2
                                  !  (move 0 to cef)
    or     %g0 0, %g5
                                     !  (jump 13155)
    ba     L13155
    nop
                                     !  (make-label)


L13147:


                                    !  (move 0 to global)
    or    %g0, 0, %g1
                                    !  (store global in cp 0 commits)
    set   (_DSpace + 2000) , %o0
    st    (%g1, [%o0 + %g4]
                                    !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                    !  (move difference sp 24 to sp)
    sub   %g2, 24, %g2
                                    !  (move 2 to cef)
    or    %g0, 2, %g5
                                    !  (jump 13319)
    ba    L13319
    nop
                                    !  (make-label)                     !!  then


L13153:


                                    !  (move (reg 0) to (reg 2))
    or    %g0, %r28, %r26
                                    !  (jump 13235)
    ba    L13235
    nop
                                    !make-label                         !!  the given


L13155:


                                                                        !!  (truth-value-integer)
                                    !  (move 1 to (reg 5))              !!  #1
    or    %g0, 1, %r23
                                    !  (compare (reg 5) with 1)
    subcc %r23, 1, %g0
                                    !  (branchlessthan 13180)
    bneg  L13180
    nop
                                    !  (move (reg 2) to global)
    or    %g0, %r26, %g1
                                    !  (move (reg 5) to arg)
    or    %g0, %r23, %g7
                                    !  (make-label)


L13160:


                                    !  (compare arg with 1)
    subcc %g7, 1, %g0
                                    !  (branchequal 13167)
    be    L13167
    nop
                                      !  (load global 1 heap into global)
    set   (_DSpace + 10004), %o0
    Id    [%o0 + %g1], %g1
                                      !  (compare global with -4)
    subcc %g1, -4, %g0
                                      !  (branchequal 13180)
    be    L13180
    nop
                                      !  (move difference arg 1 to arg)
    sub   %g7, 1, %g7
                                      !  (jump 13160)
    ba    L13160
    nop
                                      !  (make-label)

L13167:

                                      !  (load global 0 heap into (reg 6))
    set   (_DSpace + 10000) , %o0
    Id    [%o0 + %g1] , %r22
                                      !  (store (reg 6) in hp 2 heap)        !!  give
    set   (_DSpace + 10008), %o0
    st    %r22, [%o0 + %g3]
                                      !  (move -4 to global)
    or    %g0, -4, %g1
                                      !  (stor e global in hp 1 heap)
    set   (_DSpace + 10004), %o0
    st    %g1, [%o0 + %g3]
                                      !  (store hp in hp 3 heap)
    set   (_DSpace + 10012), %o0
    st    %g3, [%o0 + %g3]
                                      !  (move sum hp 8 to (reg 5))
    add   %g3, 8, %r23
                                      !  (move sum hp 16 to hp)
    add   %g3, 16, %g3
                                      !  (move 0 to global)
    or    %g0, 0, %g1
                                      !  (store global in cp 0 commits)
    set   (_DSpace + 2000), %o0
    st    %g1, [%o0 + %g4]
                                      !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                      !  (move difference sp 0 to sp)
    sub   %g2, 0, %g2
                                      !  (move 0 to cef)
    or    %g0, 0, %g5
                                      !  (jump 13186)
    ba    L13186
    nop
                                      !  (make-label)

L13180:

                                      !  (move 0 to global)
    or    %g0, 0, %g1
```

```
                                        !  (store global in cp 0 commits)
    set   (_DSpace + 2000), %o0
    st    %g1, [%o0 + %g4]
                                        !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                        !  (move difference sp 0 to sp)
    sub   %g2, 0, %g2
                                        !  (move 2 to cef)
    or    %g0, 2, %g5
                                        !  (jump 13190)
    ba    L13190
    nop                                 !  (make-label)                      !!  or

L13186:

                                        !  (move (reg 5) to (reg2))
    or    %g0, %r23, %r26
                                        !  (jump 13217)
    ba    L13217
    nop                                 !  (make-label)

L13188:

                                        !  (move (reg 2) to (reg 5))
    or    %g0, %r26, %r26
                                        !  (jump 13223)
    ba    L13223
    nop
                                        !  (make-label)

L13190:

                                        !  (load cp -1 commits into global)
    set   (_DSpace + 1996), %o0
    ld    [%o0 + %g4], %g1
                                        !  (compare global with 1)
    subcc %g1, 1, %g0
                                        !  (branchequal 13229)
    be    L13229
    nop
                                        !  (make-label)


L13193:

                                                                             !!  give the
                                        !  (move 0 to global)                !!  (truth-value|integer)
    or    %g0, 0, %g1                                                        !!   stored in
                                        !  (store global in cp 0 commits)    !!   the given cell #1
    set   (_Dspace + 20000), %o0
    st    %g1, [%o0 + %g4]
                                        !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                        !  (move difference sp 24 to sp)
    sub   %g2, 24, %g2
```

```
                                        !  (move 2 to cef)
    or    %g0, 2, %g5
                                        !  (jump 13211)
    ba    L13211
    nop
                                        !  (make-label)                          !!  combine (or)

L13199:

                                        !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                        !  (load cp 0 commits into global)
    set   (_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                        !  (compare global with 0)
    subcc %g1, 0, %g0
                                        !  (branchequal 13204)
    be    L13204
    nop
                                        !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                        !  (make-label)

L13204:

                                        !  (jump 13217)
    ba    L13217
    nop
                                        !  (make-label)

L13205:

                                        !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                        !  (load cp 0 commits into global)
    set   texttt(_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                        !  (compare global with 0)
    subcc %g1, 0, %g0
                                        !  (branchequal 13210)
    be    L13210
    nop
                                        !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                        !  (make-label)

L13210:

                                        !  (jump 13223)
    ba    L13223
    nop
                                        !  (make-label)

L13211:
```

```
                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)
    set   (_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)
    subcc %g1, 0, %g0
                                      !  (branchequal 13216)
    be    L13216
    nop
                                      !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                      !  (make-label)

L13216:

                                      !  (jump 13229)
    ba    L13229
    nop
                                      !  (make-label)                        !!combine (then)

L13217:

                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)
    set   (_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)
    subcc %g1, 0, %g0
                                      !  (branchequal 13222)
    be    L13222
    nop
                                      !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                      !  (make-label)

L13222:

                                      !  (jump 13237)
    ba    L13237
    nop
                                      !  (make-label)

L13223:

                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)
    set   (_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)
    subcc %g1, 0, %g0
```

```
                                      !  (branchequal 13228)
    be    L13228
    nop
                                      !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                      !  (make-label)

L13228:

                                      !  (jump 13235)
    ba    L13235
    nop
                                      !  (make-label)

L13229:

                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)
    set   (_Dspace + 20000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)
    subcc %g1, 0, %g0
                                      !  (branchequal 13234)
    be    L13234
    nop
                                      !  (store global in cp -1 commits)
    set   (_DSpace + 1996), %o0
    st    %g1, [%o0 + %g4]
                                      !  (make-label)

L13234:

                                      !  (jump 13319)
    ba    L13235
    nop
                                      !  (make-label)                  !!  then

L13235:

                                      !  (move (reg 2) to (reg 2))
    or    %g0, %r26, %r26
                                      !  (jump 13313)
    ba    L13313
    nop
                                      !  (make-label)                  !!  it

L13237:

                                      !  (move 1 to (reg 5))
    or    %g0, 1, %r23
                                      !  (compare (reg 5) with 1)
    subcc %r23, 1, %g0
                                      !  (branchlesstha 13265)
    bneg  L13265
    nop
                                    !  (move (reg 2) to global)
    or    %g0, %r26, %g1
                                    !  (move (reg 5) to arg)
    or    %g0, %r23 + %g7
                                    !  (make-label)

L13242:

                                    !  (compare arg with 1)
    subcc %g7, 1, %g0
                                    !  (branchequal 13249)
    be    L13249
    nop
                                    !  (load global 1 heap into global)
    set   (_DSpace + 10004), %go0
    ld    [%o0 + %g1], %g1
                                    !  (compare global with -4)
    subcc %g1, -4, %g0
                                    !  (branchequal 13265)
    be    L13265
    nop
                                    !  (move difference arg 1 to arg)
    sub   %g7, 1, %g7
                                    !  (jump 13424)
    ba    L13242
                                    !  (branchequal 13265)
    nop
                                    !  (make-label)

L13249:

                                    !  (load global 0 heap into (reg 6))
    set   (_DSpace + 10000), %o0
    ld    [%o0 + %g1], %g1
                                    !  (move 0 to global                  !!  (brance-send)
    or    %g0, 0, %g1
                                    !  (load global 0 output into arg)
    set   (_Dspace + 0), %o0
    ld    [%o0 + %g1], %g7
                                    !  (move sum arg (a 1) to arg)
    add   %g7, 4, %g7
                                    !  (store arg in global 0 output)
    set   (_Dspace + 0), %o0
    st    %g7, [%o0 + %g1]
                                    !  (store (reg 6) in arg 0 output)
    set   (_Dspace + 0), %o0
    st    %r22, [%o0 + %g7]
                                    !  (move -4 to global)
    or    %g0, -4, %g1
                                    !  (store global in hp 1 heap)
    set   (_Dspace + 10004), %o0
    st    %g1, [%o0 + %g3]
                                    !  (move hp to (reg 2))
    or    %g0, %g3 + %r26
```

```
                                        !  (move sum hp 8 to hp)
    add   %g3, 8, %g3
                                        !  (move 1 to global)
    or    %g0, 1, %g1
                                        !  (store global in cp 0 commits)
    set   (_Dspace + 2000), %o0
    st    %g1, [%o0 + %g4]
                                        !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                        !  (move difference sp 0 to sp)
    sub   %g2, 0, %g2
                                        !  (move 0 to cef)
    or    %g0, 0, %g5
                                        !  (jump 13271)
    ba    L13271
    nop
                                        !  (make-label)

L13265:

                                        !  (move 0 to global)
    or    %g0, 0, %g1
                                        !  (store global in cp 0 commits)
    set   (_Dspace + 2000), %o0
    st    %g1, [%o0 + %g4]
                                        !  (move sum cp 4 to cp)
    add   %g4, 4, %g4
                                        !  (move difference sp 24 to sp)
    sub   %g2, 24, %g2
                                        !  (move 2 to cef)
    or    %g0, 2, %g5
                                        !  (jump 13283)
    ba    L13283
    nop
                                        !  (make-label)                        !!  combine (then)

L13271:

                                        !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                        !  (load cp 0 commits into global)
    set   (_Dspace + 2000), %o0
    ld    [%o0 + %g4], %g1
                                        !  (compare global with 0)
    subcc %g1, 0, %g0
                                        !  (branchequal 13276)
    be    L13276
    nop
                                        !  (store global in cp -1 commits)
    set   (_Dspace + 1996), %o0
    st    %g1, [%o0 + %g3]
                                        !  (make-label)

L13276:

                                        !  (jump 13289)
    ba    L13289
    nop
                                      !  (make-label)


L13277:


                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)

    set   (_Dspace + 2000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)

    subcc %g1, 0, %g0
                                      !  (branchequal 13282)

    be    L13282
    nop
                                      !  (store global in cp -1 commits)

    set   (_Dspace + 1996), %o0
    st    %g1, [%o0 + %g3]
                                      !  (make-label)


L13282:


                                      !  (jump 13313)
    ba    L13313
    nop
                                      !  (make-label)


L13283:


                                      !  (move difference cp 4 to cp)
    sub   %g4, 4, %g4
                                      !  (load cp 0 commits into global)

    set   (_Dspace + 2000), %o0
    ld    [%o0 + %g4], %g1
                                      !  (compare global with 0)

    subcc %g1, 0, %g0
                                      !  (branchequal 13288)

    be    L13288
    nop
                                      !  (store global in cp -1 commits)

    set   (_Dspace + 1996), %o0
    st    %g1, [%o0 + %g3]
                                      !  (make-label)


L13282:


                                      !  (jump 13319)
    ba    L13319
    nop
                                      !  (make-label)


L13289:
```

Appendix I

Mini-Ada Action Semantics

I.1 Abstract Syntax

grammar:

(1)	Program	= ⟦ Declarations Identifier ⟧ .
(2)	Declarations	= ⟦ Declarations Declarations ⟧ |
		⟦ Identifier “:” “constant” “:=” Expression “;” ⟧ |
		⟦ Identifier “:” Nominator “;” ⟧ |
		⟦ Identifier “:” Nominator “:=” Expression “;” ⟧ |
		⟦ “type” Identifier “is” “array”
		“(” “0” “..” Expression “)” “of” Primitive “;” ⟧ |
		⟦ “function” Identifier “return” “integer” “is” Block “;” ⟧ |
		⟦ “function” Identifier “(” Formals-In “)”
		“return” “integer” “is” Block “;” ⟧ |
		⟦ “procedure” Identifier “is” Block “;” ⟧ |
		⟦ “procedure” Identifier “(” Formals “)” “is” Block “;” ⟧ .
(3)	Formals	= ⟦ Formal “;” Formals ⟧ | Formal .
(4)	Formal	= ⟦ Identifier “:” “in” “Out” “integer” ⟧ .
(5)	Formals-In	= ⟦ Formal-In “;” Formals-In ⟧ | Formal-In .
(6)	Formal-In	= ⟦ Identifier “:” “Integer” ⟧ .
(7)	Nominator	= Primitive | Identifier .
(8)	Primitive	= “boolean” | ‘integer” .
(9)	Statements	= ⟦ Statements Statements ⟧ |

```
                                ⟦ “null” “;” ⟧ |
                                ⟦ Name “:=” Expression “;”⟧ |
                                ⟦ “if” Expression “then” Statements “end” “if” “;” ⟧ |
                                ⟦ “if” Expression “then” Statements
                                “else” Statements “end” “if” “;” ⟧ |
                                ⟦ “select” Alternatives “end” “select” “;”⟧ |
                                ⟦ “select” Alternatives “else” Statements “end” “select” “;” ⟧ |
                                ⟦ “loop” Statements “end” “loop” “;”⟧ |
                                ⟦ “while” Expression “loop” Statements “end” “loop” “;”⟧ |
                                ⟦ “exit” “;”⟧ |
                                ⟦ “begin” Statements “end ” “;” ⟧ |
                                ⟦ “declare” Declarations “begin” Statements “end” “;” ⟧ |
                                ⟦ Identifier “;”⟧ |
                                ⟦ Identifier “(” Names “)” “;” ⟧ |
                                ⟦ “return” “;”⟧ |
                                ⟦ “return” Expression “;”⟧ |
                                ⟦ “write “Expression “;” ⟧ |
                                ⟦ “read” Name “;” ⟧ .
(10)  Block                   = ⟦ “begin” Statements “end” ⟧ |
                                ⟦ Declarations “begin” Statements “end” ⟧ .
(11)  Alternatives            = Statements |
                                ⟦ “when” Expression “=>” Statements ⟧ |
                                ⟦ Alternatives “or” Alternatives ⟧ .
(12)  Names                   = Name | ⟦ Names “;” Names ⟧ .
(13)  Name                    = Identifier | ⟦ Identifier “(” Expressions “)” ⟧ .
(14)  Expressions             = Expression | ⟦ Expressions “;” Expressions ⟧ .
(15)  Expressions             = “true” | “false” | Integer | Name | .
                                ⟦ “(” Expression “)” ⟧ | .
                                ⟦ “not” Expression ⟧ | .
                                ⟦ Expression Binary-Operator Expression ⟧ | .
                                ⟦ Expression Control-Operator Expression ⟧ .
(16)  Binary-Operator         = “+” | “−” | “=” | “/ =” | “<” |“<=” | .
                                “>” | “>=” | “and” | “or” | “xor” .
(17)  Control-Operator        = ⟦ “and” “then” ⟧ | ⟦ “or” “else” ⟧ .
(18)  Integer                 = natural | ⟦ “−” natural ⟧ .
```

(19) Identifier = token .

I.2 Semantic Entities

I.2.1 Items

introduces: item , parameter-less-procedure , parameterized-procedure , parameter-less-function , parameterized-function , non-abstraction , escape-reason , exit , function-return , procedure-return , there-is-given-an-exit , there-is-given-a-return , there-is-given-a-procedure-return , err .

(1) item = truth-value | integer .

(2) parameter-less-procedure = abstraction .

(3) parameterized-procedure = abstraction .

(4) parameter-less-function = abstraction .

(5) parameterized-function = abstraction .

(6) non-abstraction = item | cell | list .

(7) escape-reason = [integer] list .

(8) exit = list of 0 .

(9) function-return = [integer] list .

(10) procedure-return = list of 2 .

(11) there-is-given-an-exit = (component# 1 items it) is 0 .

(12) there-is-given-a-return =
either((component# 1 items it) is 1 , (component# 1 items it) is 2) .

(13) there-is-given-a-procedure-return = (component# 1 items it) is 2 .

(14) err = commit and then fail .

I.2.2 Closures

introduces: function-return-of _ , returned-value-of _ , parameter-less-closure _ , parameterized-function-closure _ , parameterized-procedure-closure _ .

- function-return-of _ :: integer → [integer] list .
- returned-value-of _ :: [integer] list → integer .
- parameter-less-closure _ :: act → dependent datum .
- parameterized-function-closure _ :: act → dependent datum .
- parameterized-procedure-closure _ :: act → dependent datum .

(1) function-return-of i:integer = concatenation(list of 1, list of i) .

(2) returned-value-of l:[integer] list = component# 2 items l) .

(3) parameter-less-closure A:act = closure abstraction of A & [perhaps using ()] act .

(4) parameterized-function-closure A:act =
closure abstraction of A & [perhaps using [integer] list] act .

(5) parameterized-procedure-closure A:act =
closure abstraction of A & [perhaps using [integer cell] list] act .

I.3 Semantic Functions

introduces: run _ , elaborate _ , actualize-formals _ , actualize-formal _ , ,
actualize-formals-in _ , actualize-formal-in _ ,
allocate-for _ , allocate-for-primitive _
execute _ , execute-block _ , exhaust _ ,
multi-investigate _ , investigate _
multi-evaluate _ , evaluate _ ,
the-binary-operation-result-of _
the-control-operation-completion-of _
integer-value _ , id _ .

I.3.1 Program

- run _ :: Program → act .

```
(1) run ⟦D:Declarations I:Identifier ⟧ =
        | furthermore elaborate D
        hence
        | enact application (the parameter-less-procedure bound to id I) to () .
```

I.3.2 Declarations

```
•   elaborate _ :: Declarations → act .
(1) elaborate ⟦ $D_1$:Declarations $D_2$:Declarations ⟧ = elaborate $D_1$ before elaborate $D_2$ .
(2) elaborate ⟦ $I$:Identifier ":" "constant" ":=" $E$:Expression ";" ⟧ =
        evaluate $E$ then bind id $I$ to it .
(3) elaborate ⟦ I:Identifier ":" N:Nominator ";" ⟧ =
        allocate-for $N$ then bind id $I$ to it .
(4) elaborate ⟦ $I$:Identifier ":" $N$:Nominator ":=" $E$:Expression ";" ⟧ =
        | allocate-for $N$ and then evaluate $E$
        then
        | store the given item #2 in the given cell #1 and then
        | bind id $I$ to the given datum #1 .
(5) elaborate ⟦ "type" $I$:Identifier "is" "array"
        "(" "0" ".." $E$:Expression ")" "of" "boolean" ";" ⟧ =
        bind id $I$ to parameter-less-closure
        | | give empty-list & [truth-value cell] list and then
        | | | evaluate $E$ then give sum(it, 1)
        | then
        | | unfolding
        | | | | check the given integer #2 is 0 and then
        | | | | give the given list #1
        | | | or
        | | | | | regive and then
        | | | | | allocate truth-value cell
        | | | | then
        | | | | | | give concatenation(list of the given truth-value cell #3, the given list #1)
        | | | | | and then
        | | | | | | give difference(the given integer #2, 1)
        | | | | then
        | | | | | unfold .

(6) elaborate ⟦ "type" $I$:Identifier "is" "array"
        "(" "0" ".." $E$:Expression ")" "of" "integer" ";" ⟧ =
        bind id $I$ to parameter-less-closure
        give empty-list & [integer cell] list and then
          evaluate $E$ then give sum(it, 1)
      then
        unfolding
            check the given integer #2 is 0 and then
            give the given list #1
          or
              regive and then
              allocate integer cell
            then
                give concatenation(list of the given integer cell #3, the given list #1)
              and then
                give difference(the given integer #2, 1)
            then
              unfold .
(7)  elaborate ⟦ “function” $I$:Identifier “return” “integer” ‘is” $B$:Block “;” ⟧ =
    bind id $I$ to parameter-less-closure
        execute-block $B$ and then err
      trap give returned-value-of the given function-return #1 .
(8)  elaborate ⟦ “function” $I$:Identifier “(” $F$:Formals-In “)”
    “return” “integer” “is” $B$:Block “;” ⟧ =
    bind id $I$ to parameter-function-closure
      furthermore actualize-formals-in $F$ thence
          execute-block $B$ and then err
        trap give returned-value-of the given function-return #1 .
(9)  elaborate ⟦ “procudure” $I$:Identifier “is” $B$:Block “:” ⟧ =
    bind id $I$ to parameter-less-closure
      execute-block $B$
      trap check there-is-given-a-procedure-return .
(10) elaborate ⟦ “procedure” $I$:Identifier “(” $F$:Formals “)” ‘is” $B$:Block “;” ⟧ =
    bind id $I$ to parameterized-procedure-closure
      furthermore actualize-formals $F$ thence
        execute-block $B$
        trap check there-is-given-a-procedure-return .
```

I.3.3 Formals

- actualize-formals _ :: Formals → act .

(1) actualize-formals ⟦ F_1:Formal ";" F_2:Formals ⟧ =
```
    | give head the given list #1 then actualize-formal $F_1$
    before
    | give tail the given list #1 then actualize-formals $F_2$ .
```

(2) actualize-formals F:Formal =
give head the given list #1 then actualize-formal F .

I.3.4 Formal

- actualize-formal _ :: Formal → act .

(1) actualize-formal ⟦ I:Identifier ":" "in" "out" "integer" ⟧ =
bind id I to the given integer cell #1 .

I.3.5 Formals-In

- actualize-formals-in _ :: Formals-In → act .

(1) actualize-formals-in ⟦ F_1:Formal-In ";" F_2:Formals-In ⟧ =
```
    | give head the given list #1 then actualize-formal-in $F_1$
    before
    | give tail the given list #1 then actualize-formals-in $F_2$ .
```

(2) actualize-formals-in F:Formal-In =
give head the given list #1 then actualize-formal-in F .

I.3.6 Formal-In

- actualize-formal-in _ :: Formal-In → act .

(1) actualize-formal-in ⟦ I:Identifier ":" "integer" ⟧ =
bind id I to the given integer #1 .

I.3.7 Nominator

- allocate-for _ :: Nominator → act .

(1) allocate-for P:Primitive = allocate-for-primitive P .

(2) allocate-for I:Identifier =
    enact application (the abstraction bound to id I) to () .

I.3.8 Primitive

- allocate-for-primitive _ :: Primitive → act .

(1) allocate-for-primitive “boolean” = allocate truth-value cell .

(2) allocate-for-primitive “Integer = allocate integer cell .

I.3.9 Statements

- execute _ :: Statements → act .

(1) execute ⟦ S_1:Statements S_2:Statements ⟧ = execute S_1 and then execute S_2 .

(2) execute ⟦ “null” “;” ⟧ = complete .

(3) execute ⟦ N:Name “:=” E:Expression ⟧ =

```
    | investigate $N$ and then evaluate $E$
    then store the given item #2 in the given cell #1 .
```

(4) execute ⟦ “if” E:Expression “then” S:Statements “end” “if” “;” ⟧ =

```
    evaluate $E$ then
    | | check (it is true) and then execute $S$
    | or
    | | check (it is false) .
```

(5) execute

```
    ⟦ “if” $E$:Expression “then” $S_1$:Statements “else” $S_2$:Statements “end” “if” “;” ⟧ =
    evaluate $E$ then
    | | check (it is true) and then execute $S_1$
    | or
    | | check (it is false) and then execute $S_2$ .
```

(6) execute ⟦ “select” A:Alternatives “end” “select” “;” ⟧ =

```
        | exhaust $A$
        trap
        | enact application the given abstraction #1 to ()
(7) execute ⟦ “select” $A$:Alternatives “else” $S$:Statements “end” “select” “;” ⟧ =
        | exhaust $A$ and then
        | | | give parameter-less-closure execute $S$
        | | then escape
        trap
        | enact application the given abstraction #1 to ()
(8) execute ⟦ “loop” $S$:Statements “end” “loop” “;” ⟧ =
        | unfolding
        | | execute $S$ and then unfold
        trap
        | | chech there-is-given-an-exit
        | or
        | | check there-is-given-a-return and then escape .
(9) execute ⟦ “while” $E$:Expression “loop” $S$:Statements “end” “loop” “;” ⟧ =
        | unfolding
        | | evaluate $E$ then
        | | | | check (it is true) and then execute $S$ and then unfold
        | | | or check (it is false)
        trap
        | | check there-is-given-an-exit
        | or
        | | check there-is-given-an-return and then escape .
(10) execute ⟦ “exit” “;” ⟧ = give exit then escape .
(11) execute ⟦ “begin” $S$:Statement “end” “;” ⟧ = execute $S$ .
(12) execute ⟦ “declare” $D$:Declarations “begin” $S$:Statements “end” “;” ⟧ =
        furthermore elaborate $D$ hence
        | execute $S$ .
(13) execute ⟦ $I$:Identifier “;” ⟧ =
        enact application the parameter-less-procedure bound to id $I$ to () .
(14) execute ⟦ $I$:Identifier “(” $N$:Names “)” “;” ⟧ =
        | give the parameterized-procedure bound to id $I$ and then
        | multi-investigate $N$
        then
        | enact application the given abstraction #1 to the given list #2 .
```
(15) execute ⟦ "return" ";" ⟧ = give procedure-return then escape .

(16) execute ⟦ "return" E:Expression ";" ⟧ =
```
          evaluate $E$ then
          give function-return-of it then
          escape .
```
(17) execute ⟦ "write" E:Expression ";" ⟧ =
```
          evaluate $E$ then batch-send it .
```
(18) execute ⟦ "read" N:Names ";" ⟧ =
```
          | batch-receive an integer and then investigate $N$
          then store the given integer #1 in the given integer cell #2 .
```

I.3.10 Block

- execute-block _ :: Block → act .

(1) execute-block ⟦ "begin" S:Statements "end" ⟧ = execute S .

(2) execute-block ⟦ D:Declarations "begin" S:Statements "end" ⟧ =
```
          furthermore elaborate $D$ hence
          | execute $S$ .
```

I.3.11 Alternatives

- exhaust _ :: Alternatives → act .

(1) exhaust S:Statements =
```
          | give parameter-less-closure execute $S$
          then escape .
```
(2) exhaust ⟦ "when" E:Expression "=>" S:Statements "end" ⟧ =
```
          evaluate $E$ then
          | | | check (it is true) then
          | | | give parameter-less-closure execute $S$
          | | then escape
          | or check (it is false) .
```

(3) exhaust ⟦ A_1:Alternatives "or" A_2:Alternatives ⟧ =
```
        exhaust $A_1$ and then exhaust $A_2$ .
```

I.3.12 Names

- multi-investigate _ :: Names → act .

(1) multi-investigate N:Name =
```
        investigate $N$ then give list of it .
```

(2) multi-investigate ⟦ N_1:Names ";" N_2:Names ⟧ =
```
        | multi-investigate $N_1$ and then multi-investigate $N_2$
        then give concatenation(the given list #1, the given list #2) .
```

I.3.13 Name

- investigate _ :: Name → act .

(1) investigate I:Identifier =
```
        give the datum bound to id $I$ then
        | give the given non-abstraction #1 or
        | enact application the given parameter-less-function #1 to () .
```

(2) investigate ⟦ I:Identifier "(" E:Expressions ")" ⟧ =
```
        | give the datum bound to id $I$ and then
        | multi-evaluate $E$
        then
        | | | give the given list #1 and then
        | | | give head the given [integer] list #2
        | | then
        | | | give component# sum(the given integer #2, 1) items
        | | | (the given list #1)
        | or
        | | enact application (the given parameterized-function #1)
        | | to (the given list #2) .
```

I.3.14 Expressions

- multi-evaluate _ :: Expressions → act .

(1) multi-evaluate E:Expression = evaluate E then give list of it .

```
(2) multi-evaluate ⟦ $E_1$:Expressions ";" $E_2$:Expressions ⟧ =
        | multi-evaluate $E_1$ and then multi-evaluate $E_2$
        then give concatenation(the given list #1, the given list #2) .
```

I.3.15 Expression

```
•   evaluate _ :: Expressions → act .
(1) evaluate "true" = give true .
(2) evaluate "false" = give false .
(3) evaluate $i$:Integer = give integer-value $i$ .
(4) evaluate $N$:Name =
        investigate $N$ then
        | give the given item #1 or
        | give the item stored in the given cell #1 .
(5) evaluate ⟦ "(" $E$:Expression ")" ⟧ = evaluate $E$ .
(6) evaluate ⟦ "not" $E$:Expression ⟧ = evaluate $E$ then give not it .
(7) evaluate ⟦ $E_1$:Expression $O$:Binary-Operator $E_2$:Expression ⟧ =
        | evaluate $E_1$ and then evaluate $E_2$
        then give the-binary-operation-result-of $O$ .
(8) evaluate ⟦ $E_1$:Expression $O$:Control-Operator $E_2$:Expression ⟧ =
        evaluate $E_1$ then
        | | check the-control-operation-completion-of $O$ and then
        | | give the given truth-value #1
        | or
        | | check not the-control-operation-completion-of $O$ and then
        | | evaluate $E_2$ .
```

I.3.16 Binary-Operator

```
•   the-control-operation-completion-of _ :: Control-Operator → dependent datum .
(1) the-binary-operation-result-of "+" =
        sum(the given integer #1 the given integer #2) .
(2) the-binary-operation-result-of "–" =
        difference(the given integer #1, the given integer #2) .
```

(3) the-binary-operation-result-of “=” =
(the given intem #1) is (the given intem #2) .

(4) the-binary-operation-result-of “/ =” =
not((the given intem #1) is (the given intem #2)) .

(5) the-binary-operation-result-of “<” =
(the given integer #1) is less than (the given integer #2) .

(6) the-binary-operation-result-of “<=” =
not((the given integer #2) is less than (the given integer #1)) .

(7) the-binary-operation-result-of “>” =
(the given integer #2) is less than (the given integer #1) .

(8) the-binary-operation-result-of “>=” =
not ((the given integer #1) is less than (the given integer #2)) .

(9) the-binary-operation-result-of “and” =
both(the given integer #1, the given integer #2) .

(10) the-binary-operation-result-of “or” =
either(the given truth-value #1, the given truth-value #2) .

(11) the-binary-operation-result-of “xor” =
not((the given truth-value #1) is (the given truth-value #2)) .

I.3.17 Control-Operator

- the-control-operation-completion-of _ :: Control-Operator $\rightarrow$ dependent datum .

(1) the-control-operation-completion-of $[\![$ “and” “then” $]\!]$ =
(the given truth-value #1) is false .

(2) the-control-operation-completion-of $[\![$ “or” “else” $]\!]$ =
(the given truth-value #1) is true .

I.3.18 Integer

- integer-value _ :: Integer $\rightarrow$ integer .

(1) integer-value n:natural = n .

(2) integer-value $[\![$ “$-$” n:natural $]\!]$ = negation n .

I.3.19 Identifier

- id _ :: Identifier $\rightarrow$ token .

(1) id k:token $= k$.

Appendix J

Mini-Ada Benchmark Programs

```
/*
 * This Mini-Ada program performs a sorting exercise for timing purposes.
 *
 * input:  an integer n, n ≤ 1000
 * action: sorts 0..n−1 into descending order using bubble-sort
 * output:original array: ..n−1
 *         marker        : −999
 * sorted  array         : n−1..0
*/

max_size : constant := 1000 ;
type numtype is array ( 0 .. 1000 ) of integer ;
num : numtype ;   /* The array to sort */


procedure sort ( numElts : in out integer ) is
/*
 * This procedure uses a bubble sort to arrange the 0..numElts−1
 * elements of the “num” array
 */
    last : integer ;
    current : integer ;
    temp : integer ;
    begin
        last : = numElts −1 ;
        while 0 < last loop
            current : = 0 ;

            while current < last loop
                if  num ( current ) < num ( current +1 ) then
                    temp : = num ( current ) ;
                    num ( current ) := num(current +1) ;
                    num ( current +1 ) := temp ;
                else
                    null ;
                end if ;
                current : = current +1 ;
            end loop ;

            last = last −1 ;
        end loop ;
    end ; /* sort */


procedure printNums ( size : in out integer ) is
/*
 *  Print out the "num" array, 0..size−1
 */
    i : integer ;

    begin
        i := 0 ;
        while i < size loop
            write num ( i ) ;
            i := i +1 ;
        end loop ;
    end ; /* printNums */
```

```
/*
 *  Main program
 */

i : integer ;
numSort : integer ;   /* The number of integers to sort */


procedure main is
    begin
        read numSort ;

        if maxsize < numsort
        then null ;
        else
            i := 0 ;

            while  i < numSort  loop
                num(i) := i ;
                i := i +1 ;
            end loop ;

            printNums ( numSort ) ;
            write − 999 ;
            sort ( numSort ) ;
            printNums ( numSort ) ;
        end if ;
    end ;

main

/*
* This Mini-Ada program performs the sieve of Erathosthenes prime number
* generator. There is neither input nor output; the program is solely used
* for timing purposes.
*/
```

```
type arr is array ( 0 .. 512 ) of boolean ;
flags : arr ;
size : constant := 512 ;
iterations : constant := 1 ;

procedure primes is
    i : integer ;
    prime : integer ;
    j : integer ;
    count : integer ;
    loops : integer ;
    begin
        loops := 0 ;
        while loops < iterations loop
            count := 0 ;
            i := 0 ;
            while i < size loop
                flags ( i ) := true ;
                i := i +1 ;
            end loop ;
            i := 0 ;
            while i < size loop
                if flags ( i ) then
                    prime := i + i +3 ;
                    j := i + prime ;
                    while j < size loop
                        flags ( j ) : = false ;
                        j := j + prime ;
                    end loop ;
                    count :=count +1 ;
                end if ;
                i := i +1 ;
            end loop ;
            loops := loops +1 ;
        end loop ;
    end ;
```

primes

```
/*
 * This Mini-Ada program performs Euclid's algorithm 30 times.
 *
 * input:   three integers i, j, k
 * action:  performs k times Euclid's algorithm on i and j
 * output:  the greatest common divisor of i and j, k times
*/

procedure euclid ( i : in out integer ; j : in out integer ) is
    begin
        while i / = j loop
            if i < j
            then j := j − i ;
            else i := i − j ;
            end if ;
        end loop ;
        write i ;
    end ;

procedure run is
    i : integer ; j : integer ;
    i2 : integer ; j2 : integer ; k : integer ;
    begin
        read i ; read j ;
        read k ;
        while k > 0 loop
            i2 := i ;
            j2 := j ;
            euclid ( i2 ; j2 ) ;
            k := k −1 ;
        end loop ;
    end ;

run
```

```
/*
 * This Mini-Ada program computes the n'th fibonacci number 36 times.
 *
 * input:   an integer n
 * action:  computes the n'th fibonacci number iteratively
 * output:  the n'th fibonacci number
*/

procedure fibonacci ( n : in out integer ) is
    i : integer ; j : integer ;
    begin
        select
            when n < 0 => write −999 ;
        or
            when n = 0 or n = 1 => write 1 ;
        else
            n := n −1 ;
            i := 1 ;
            j := 1 ;
            while n / = 0 loop
                declare k : integer ;
                begin
                    k := i ; i := i + j ; j := k ;
                end ;
                n := n −1 ;
            end loop ;
            write i ;
        end select ;
    end ;
```

```
procedure run is
    n : integer ; k : integer ;
    begin
        read n ;
        k := 36 ;
        while k > 0 loop
            fibonacci ( n ) ;
            k := k −1 ;
        end loop ;
    end ;

run
```

Appendix K

Informal Summary of Action Notation

K.1 Basic Action Notation

Basic action notation is primarily concerned with specifying flow of control in performances of actions. It includes basic notation for data as well.

K.1.1 Actions

The notation for specifying actions consists of action *primitives*, which may involve dependent data, and action *combinators*, which operate on one or two *subactions*. There is also notation for specifying *sorts* of actions, as follows:

- 'act': the sort of all actions.
- '[perhaps using D_0] act': a sort of action, where D_0 is a (sort of) dependent data. Restricts 'act' to those actions which, when performed, perhaps evaluate dependent data that refers to the current information indicated by D_0.
- Primitive basic actions:
  - Give no data, except for 'escape'.
  - Produce no bindings.

  - Make no changes to storage.
  - Do not communicate.

- ‘complete’: a primitive basic action. Represents normal termination. Unit for ‘_ and _’, as well as for ‘_ and then _’.
  - Indivisible. Always completes.

- ‘escape’: a primitive basic action. Represents abnormal termination. Unit for ‘_ trap _’.
  - Indivisible. Always escapes.
  - Gives any given data.

- ‘fail’: a primitive basic action. Represents abortion of the current alternative. Unit for ‘_ or _’.
  - Indivisible. Always fails.

- ‘commit’: a primitive basic action. Represents commitment to the current alternative, cutting away other current alternatives.
  - Indivisible. Always commits and completes.

- ‘diverge’: a basic action. Represents nontermination.
  - Always diverges.

- ‘unfold’: a dummy action, standing for the innermost enclosing unfolding.

- ‘unfolding A’: a basic ‘combination’ of action A. Represents the (in general, infinite) action formed by continually substituting A for ‘unfold’. (To avoid singularities in pathological cases, substitute ‘complete and then A’, rather than just A.)
  - Performs A, but whenever the dummy action ‘unfold’ is reached, A is performed in place of ‘unfold’.

- Basic action combinators are:

  - Functionally conducting (see 'A_1 and A_2'), except for 'A_1 trap A_2, which is composing (see 'A_1 then A_2').
  - Declaratively conducting. (See 'A_1 and A_2'.)

- 'indivisibly A': a basic 'combination' of action A. Represents that A is not interleaved with other actions of the same agent. For use only when A cannot diverge.
  - Indivisible. A is performed as a single step.

- 'A_1 or A_2': a basic combination of actions A_1, A_2. Represents implementationdependent choice; specializes to deterministic choice when one or the other of A_1, A_2 must fail.
  - Basically alternatives. Performs either A_1 or A_2. When the performed alternative fails without committing, it is ignored and the other alternative is performed instead.
  - Functionally conducting. All the data given to the combination of A_1, A_2 is given to the performed alternative. On normal or abnormal termination, all the data given by the performed alternative is given by the combined action.
  - Declaratively conducting. All the bindings received by the combination of A_1, A_2 are received by the performed alternative. On normal termination, all the bindings produced by the performed alternative are produced by the combined action.

- 'A_1 and A_2': a basic combination of actions A_1, A_2. Represents implementationdependent order of performance of indivisible subactions, specializing to independent performance when there is no 'interference' between A_1 and A_2.
  - Basically interleaving. Performs both A_1, A_2, with arbitrary interleaving of their indivisible steps. An escape or a failure causes any remaining parts of the subactions to be skipped.
  - Functionally conducting. The data given to the combination of A_1, A_2 is given to both A_1, A_2. On normal termination, all the data given by A_1, A_2 is collected and given by the combined action—if both give one or more items of data, these are tupled in the given

order. On escape, only the data given by the escape is given by the combined action.

  - Declaratively conducting. The bindings received by the combination of A_1, A_2 are received by both A_1, A_2. On normal termination, all the bindings produced by A_1, A_2 are collected and produced by the combined action—provided that the bindings are all for distinct tokens, otherwise the combined action fails.

- 'A_1 and then A_2': a basic combination of actions A_1, A_2. Represents dependency on normal termination.

  - Basically (normal) sequencing. Performs A_1 first. If A_1 completes, performs A_2.

- 'A_1 trap A_2': a basic action combination. Represents recovery from abnormal termination.

  - Basically abnormal sequencing. Performs A_1 first. If A_1 escapes, performs A_2.
  - Functionally composing. (See 'A_1 then A_2'.)

K.1.2 Dependent Data

- 'dependent D': a sort. Its individuals are items of dependent data that, when evaluated, always yield data of sort D.

- Every *data-operation* (i.e., operation specified only for arguments included in 'data') is extended to arguments of sort 'dependent data'. The application of a data operation to dependent data yields whatever is yielded by applying the data operation to the data yielded by the arguments.

K.1.3 Data

- 'datum': a sort. Its individuals represent items of data. Left open, as it depends on the variety of information processed by the programs of a programming language. Includes generally-useful sorts from Data Notation, except for tuples. Similarly for 'distinct-datum', the sort of datum whose individuals are distinguished by the operation '_ is _'.

- ‘data’: a sort. Its individuals represent ordered collections of individuals of sort ‘datum’, processed as transient information.
- ‘a D’: the same data as D. Only used to improve the readability of the notation. Similarly for ‘an D’, ‘the D’, and ‘of D’. Thus an application of an operation *op* to arguments x can be written as ‘*op* x’, ‘*op* of x’, and ‘the *op* of x’. Note that ‘the’ and ‘of’ are obligatory parts of some other (data-) operation symbols. (Compare the HyperCard scripting language, HyperTalk [23].)

K.2 Functional Action Notation

Functional action notation is primarily concerned with specifying the processing of transient information (data).

K.2.1 Actions

- Primitive functional actions:
  - Do not commit.
  - Produce no bindings.
  - Make no changes to storage.
  - Do not communicate.
- ‘give D’: a primitive functional action, where D is dependent data. Represents creating a piece of transient information.
  - Indivisible. Completes when D yields data. Fails when D yields nothing.
  - Gives the data yielded by D.
- ‘regive’: a primitive functional action. Represents propagation of transient information, i.e., data. Unit for _ then _.
  - Indivisible. Always completes.
  - Gives any given data.

- ‘check D’: a functional action, where D is a dependent truth-value. Represents a guard checking that a condition is true.
  - Indivisible. Completes when D yields true. Fails when D yields false (or nothing).
  - Gives no data.
- Functional action combinators are:
  - Declaratively conducting. (See ‘A_1 and A_2.)
- ‘A_1 then A_2: a functional combination of actions A_1, A_2. Represents passing on transient information normally.
  - Basically sequencing. (See ‘A_1 and then A_2’.)
  - Functionally composing. The data given to the combination of A_1, A_2 is given to A_1. Only the data given by A_1 is given to A_2 (provided that A_2 is performed). Only the data given by A_2 is given by the combined action.

K.2.2 Dependent Data

- ‘given $D\#p$’ a dependent datum, where D is a sort of datum and p is a positive integer. Yields the p’th component of the data given to its evaluation, provided that the datum is of sort D.
- ‘it’: a dependent datum. Yields the single datum given to its evaluation as a transient.
- ‘them’: dependent data. Yields all the data given to its evaluation as transients.

K.2.3 Data

- ‘data’: a sort. Its individuals represent ordered collections, i.e., tuples, of individuals of sort ‘datum’, processed as transient information.

K.3 Declarative Action Notation

Declarative action notation is primarily concerned with specifying the processing of scoped information (bindings).

K.3.1 Actions

- Primitive declarative actions:
  - Do not commit.
  - Give no data.
  - Make no changes to storage.
  - Do not communicate.
- ‘bind T to D’: a primitive declarative action, where T is a token and D is a dependent data. Represents creating a piece of scoped information.
  - Indivisible. Completes when D yields a data of sort ‘bindable’. Fails otherwise.
  - Produces the binding of the token T to the data D.
- ‘furthermore A’: a declarative combination of action A. Represents propagating the received bindings, but letting bindings produced by A take precedence when there is a conflict.
  - Baically as A.
  - Functionally as A.
- ‘A_1 hence A_2’ a declarative combination of actions A_1, A_2. Represents passing on scoped information, restricting the bindings received by A_2.
  - Basically sequencing. (See ‘A_1 and then A_2’.)
  - Functionally conducting. (See ‘A_1 and A_2’.)
  - Declaratively composing. The bindings received by the combination of A_1, A_2 are received by A_1. Only the bindings produced by A_1 are received by A_2 (provided that it is performed). Only the bindings produced by A_2 are produced by the combined action.

- 'A_1 before A_2': a declarative combination of actions A_1, A_2. Represents accumulating scoped information.
  - Basically sequencing. (See 'A_1 and then A_2'.)
  - Functionally conducting. (See 'A_1 and A_2')
  - Declaratively accumulating. The bindings received by the combination of A_1, A_2 are received by A_1. The bindings received by the combined action, overlaid with the bindings produced by A_1, are received by A_2 (provided that it is performed). The bindings produced by A_1, A_2 are collected and produced by the combined action—provided that the bindings are all for distinct tokens, otherwise the action fails.

K.3.2 Dependent Data

- 'the D bound to T': dependent data, where D is a sort of data and T is a token. Yields the data of sort D to which T is bound by the received bindings, if any.

K.3.3 Data

- 'bindings': a subsort of 'map'. Its individuals represent collections of associations between tokens and bindable individuals.
- 'token': a subsort of 'distinct-datum'. Left unspecified, as it depends on the variety of identifiers of a programming language. (Usually, it is a subsort of 'string'.)
- 'bindable': a subsort of 'data'. Left open, as it depends on the variety of scoped information processed by the programs of a programming language.

K.4 Imperative Action Notation

Imperative action notation is primarily concerned with specifying the processing of stable information (storage).

K.4.1 Actions

- Primitive imperative actions:
  - Give no data.
  - Produce no bindings.
  - Do not communicate.
- store D_1 in D_2: a primitive imperative action, where D_1 is a dependent storable and D_2 is a dependent cell. Represents changing a piece of stable information.
  - Indivisible. Commits and completes when D_2 yields a reserved cell and D_1 yields a data that is storable in that dell. Fails otherwise.
  - Stores the storable yielded by D_1 in the cell yielded by D_2.

K.4.2 Dependent Data

- 'the D_1 stored in D_2': dependent data, where D_1 is a sort of data and D_2 is a dependent cell. Yields the data of sort D_1 stored in the cell yielded by D_2, according to the current storage, provided that the cell is reserved.

K.4.3 Data

- 'storage': a subsort of 'map'. Its individuals represents states of stable information, associating cells with storable (or 'uninitialized') individuals.
- 'cell': a subsort of 'distinct-datum'. Its individuals represent the locations of pieces of stable information. Left loosely-specified, as the details are implementation dependent.
- 'storable': a subsort of 'data'. Left unspecified, as it depends on the variety of stable information processed by the programs of a programming language.
- 'uninitialized': an individual datum. Represents the absence of a stored datum in a reserved cell.

K.5 Reflective Action Notation

Reflective action notation is concerned with specifying the reification of actions and information as abstractions, and with the reflection of abstractions as actions.

K.5.1 Actions

- ‘enact D’; a reflective action, where D is a dependent abstraction. Represents performing an action in a context different from that of its occurrence.
  - Performs the action incorporated in the abstraction yielded by evaluating D.
  - The performance of the incorporated action is given no data. (But see the dependent datum ‘application D_1 to D_2’.)
  - The performance of the incorporated action receives no bindings. (But see the dependent datum ‘closure D’.)

K.5.2 Dependent Data

- ‘application D_1 to D_2’: an abstraction, where D_1 is an abstraction and D_2 is data. Incorporates the same action as D_1, except that the incorporated action is given D_2 as transients whenever the abstraction is enacted. Represents supplying transients to an abstraction (precluding the supply of further transients).

  This operation extends to dependent data D_1, D_2 in the usual way: it is evaluated by applying the above operation to the data yielded by evaluating D_1, D_2.

- ‘closure D’ a dependent abstraction, where D is an abstraction. Yields an abstraction which incorporates the same action as D except that the incorporated action receives particular bindings whenever the abstraction is enacted. The bindings are those current for the evaluation of ‘closure D’.

This operation extends to dependent data D in the usual way: it is evaluated by applying the above operation to the datum yielded by evaluating D.

'closure abstractios of A' represents reification of an action as an abstraction with static bindings 'enact the closure of a given abstraction' represents reflection of an abstraction with dynamic bindings (unless static bindings were already supplied to it). 'enact a given abstraction' represents reflection of an abstraction with no bindings (unless static bindings were already supplied to it).

K.5.3 Data

- 'abstraction': the sort of datum that incorporates (i.e., reifies) an action. The incorporated action is performed when the abstraction is enacted (i.e., reflected).
- 'abstraction of A': an abstraction, where A is an action. Incorporates A. Represents (a pointer to) the 'code' implementing A. Dependent data occurring in A does *not* get evaluated when the abstraction is evaluated: it is left for evaluation during the performance of the incorporated action.

K.6 Hybrid Action Notation

Hybrid action notation is concerned with specifying the processing of a mixture of scoped and stable information.

K.6.1 Actions

- 'A_1 thence A_2': a declarative and functional combination of actions A_1, A_2. Like 'A_1 then A_2' for data and 'A_1 hence A_2 for bindings.
  - Basically sequencing. (See 'A_1 and then A_2'.)
  - Functionally composing. (See 'A_1 then A_2'.)
  - Declaratively composing. (See 'A_1 hence A_2')

- ‘allocate D’: an imperative and functional action, where D is a sort of cell. Represents implementation dependent choice and reservation of a cell.
  - Indivisible. Commits and completes when there is an unreserved cell of sort D. Fails otherwise.
  - Reserves some cell of sort D.
  - Gives the reserved cell.

Appendix L

Data Notation

L.1 Instant

L.1.1 Distinction

(1) s $= \square$
(2) _ is _ :: s, s $\longrightarrow$ truth-value (*partial, commutative*) .

L.1.2 Partial Order

(1) _ is _, _ is less than _, _ is greater than _ :: s, s $\longrightarrow$ truth-value (*partial*) .

L.2 Truth-Values

L.2.1 Basics

(1) truth-value = truth | false (*partial*) .

L.2.2 Specifics

needs: Tuples/Basics . truth-value $\leq$ component .

(1) if _ then _ else _ :: truth-value , x, $y \rightarrow x$ | y ($linear$) .

(2) not _ :: truth-value $\rightarrow$ truth-value ($total$) .

(3) both _ :: (truth-value, truth-value) $\rightarrow$ truth-value
($total, associative, commutative, idempotent, unit\ is$ true) .

(4) either _ :: (truth-value, truth-value) $\rightarrow$ truth-value
($total, associative, commutative, idempotent, unit\ is$ false) .

(5) all _ :: (truth-value* $\rightarrow$ truth-value
($total, associative, commutative, idempotent, unit\ is$ true) .

(6) any _ :: (truth-value* $\rightarrow$ truth-value
($total, associative, commutative, idempotent, unit\ is$ false) .

(7) _ is _ :: (truth-value, truth-value) $\rightarrow$ truth-value

L.3 Numbers

L.3.1 Naturals

Basics

(1) natural = 0 | positive-integer ($disjoint$) .

(2) sucessor _ :: natural $\rightarrow$ positive-integer ($total$) .

(3) 0 :: natural .

Specifics
needs: Tuples/Basics . natural $\leq$ component .

(1) 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 : natural .

(2) _0 , _1 , _2 , _3 , _4 , _5 , _6 , _7 , _8 , _9 :: natural $\rightarrow$ natural ($total$)

(3) sum _ :: (natural* $\rightarrow$ natural
($total, associative, commutative, idempotent, unit\ is$ 0) .

(4) _ is _ , _ is less than _ :: (natural, natural) → truth-value
integer, integer → truth-value (*total*(.

L.3.2 Integers

Basics

(1) integer = 0 | nonzero-integer (*disjoint*) .

(2) nonzero-integer = positive-integer | negative-integer (*disjoint*) .

(3) successor _ predecessor _ :: integer → integer (*total*) .

Specifics

needs: Tuples/Basics . integer ≤ component .

(1) negation _ :: integer → integer (*total*) .

(2) sum _ :: integer* → integer (*total, associative, communative, unit is* 0)

(3) difference _ :: (integer, integer) → integer (*total*) .
(4) _ is _ , _ is less than _ :: integer , integer → truth-value (*total*).

L.4 Characters

L.4.1 Generics

needs: Numbers/Naturals/Basics .

(1) character = □ .

L.5 Strings

L.5.1 Basics

needs: Tuples/Basics, Lists/Basics . character $\leq$ nonlist-item .

(1)	string	[character] list .
(2)	“”	string .
(3)	- ˆ -	:: (string, string) $\rightarrow$ string (*total, associative unit is* “”) .

L.6 Trees

L.6.1 Generics

(1) nonlist-leaf = □ .

L.6.2 Basics

needs: Lists/Basics . tree $\leq$ nonlist-item .

(1) tree = nonlist-leaf | [tree] list (*disjoint*) .

L.6.3 Specifics

needs: Tuples/Basics .

(1)	$[\![\ ]\!]$	: tree .
(2)	$[\![\ \_\ ]\!]$	:: tree* $\rightarrow$ tree (*total*) .

(3)	$[\![\_\ \_]\!]$	:: tree* tree* $\rightarrow$ tree (*total*) .

L.6.4 Syntax

needs: Lists/Basics .

(1)	character	$\leq$ nonlist-leaf .
(2)	syntax-tree	= string | [syntax-tree] list .

L.7 Lists

L.7.1 Generics

(1)	nonlist-item	= □ .

L.7.2 Basics

needs: Tuples/Basics . item $\leq$ component .

(1)	list	= list of item* .
(2)	item	= nonlist-item | list (*total*) .
(3)	list of _	item* $\rightarrow$ list(*total*) .

L.7.3 Specifics

needs: Tuples/Basics .

(1) [_] _ :: litem, list $\longrightarrow$ list .

(2) items _ :: list $\longrightarrow$ item* (*total*) .

(3) head _ :: list $\longrightarrow$ item (*partial*) .

(4) tail _ :: list $\longrightarrow$ list (*partial*) .

(5) empty-list :: list .

(6) concatenation _ :: list* $\longrightarrow$ list (*total, associative, unit is* empty-list) .

L.8 Sets

L.8.1 Generics

(1) nonset-element = □ .

(2) _ is _ :: nonset-element, nonset-element $\longrightarrow$ truth-value (*total*) .

L.8.2 Basics

needs: Tuples/Basics . element $\leq$ component .

(1) set = set of element* .

(2) element = nonset-element | set (*disjoint*) .

(3) set of _ = element* $\longrightarrow$ set(*total*) .

L.8.3 Specifics

needs: Tuples/Basics , Lists .

(1) [_] _ :: element, set $\longrightarrow$ set .

(2) elements _ :: set $\longrightarrow$ element* (*strict, linear*) .

(3)	empty-set	:: set .
(4)	union _	:: set* → set (*total, associative, commutative, unit is* empty-set) .
(5)	difference _	:: set, set → set (*total*) .
(6)	intersection _	:: set$^+$ → set (*total, associative, commutative, idempotent*) .
(7)	_ restricted to _ _ omitting _	:: [element] list, set → [element] list (*total*) .
(8)	_ is in _	:: element, set → truth-value (*total*) .
(9)	_ is included in _	:: set, set → truth-value (*total*) .
(10)	_ is _	:: set, set → truth-value (*total*) .

L.9 Maps

L.9.1 Generics

(1) nonmap-range = □ .

L.9.2 Basics

needs: Tuples/Basics . map ≤ component .

(1)	map	= map of (map of element to range)* .
(2)	range	= nonmap-range | map (*disjoint*) .
(3)	map of _ to _	:: element, range → map (*total*) .
(4)	empty-map	:: map .
(5)	map of _	:: map* → map (*partial, associative, commutative, unit is* empty-map) .
(6)	mapped-set _	map → set (*total*) .

L.9.3 Specifics

needs: Tuples/Basics .

(1) [_ to _] _ = element, range, map $\rightarrow$ map .

(2) _ at _ = map, element $\rightarrow$ range (*partial*) .

(3) overlay _ :: map* $\rightarrow$ map (*total, associative, idempotent, unit is* empty-map) .

(4) _ restricted to _ , _ omitting _ :: map, set $\rightarrow$ map (*total*) .

L.10 Tuples

L.10.1 Generics

(1) component $\Box$.

L.10.2 Basics

needs: Tuples/Basics . map $\leq$ component .

(1) tuple $\geq$ component .

(2) ( ) = tuple .

(3) (_ , _) :: tuple, tuple $\rightarrow$ tuple (*total, associative, unit is* ()) .

(4) _ *, _ $^+$ :: tuple $\rightarrow$ tuple .

L.10.3 Specifics

(1) count _ :: tuple $\rightarrow$ natural (*total*) .

(2) component#_ _ :: positive-integer, tuple $\rightarrow$ component (*partial*) .

Appendix M

Informal Summary of Meta Notation

Meta-notation is for specifying formal notation: what symbols are used, how they may be put together, and their intended interpretation.

Our meta-notation here supports a *unified* treatment of sorts and individuals: an individual is treated as a special case of a sort. Thus operations can be applied to sorts as well as individuals. A vacuous sort represents the lack of an individual, in particular the ‘undefined’ result of a partial operation. Sorts may be related by inclusion; sort equality is just mutual inclusion. But a sort is not determined just by the set of individuals that it includes: it has an ‘intension’, stemming from the way it is expressed. For example, the sort of those natural numbers that are in the range of the successor operation is distinct from the sort of those that have a well-defined reciprocal, even though their sets of individuals are the same.

The meta-notation provides Horn clauses and (initial) constraints—explained below—for specifying the intended interpretation of symbols. Specifications may be divided into mutually-dependent and nested modules, presented incrementally.

M.1 Vocabulary

The vocabulary of the meta-notation consists of (constant and operation) symbols, variables, titles, and special marks.

M.1.1 Symbols

Symbols are of two forms: quoted or unquoted. Quoted symbols always stand for constants. A doubly-quoted symbol may quote an arbitrary sequence of graphic characters—except that it must be properly balanced with respect to quotation marks. A singly quoted symbol may only quote a single character. (Single quotes are also used to delimit unquoted symbols when these are exhibited in informal text; the quotes are then not part of the symbols themselves.)

In unquoted symbols the character '_' indicates the positions of arguments. Symbols without '_' *always* stand for constants.

Unquoted symbols are written here in this saris-serif font. They must not contain quotation marks at all, and they must be balanced with respect to brackets ( ), [ ], etc. A symbol may not consist entirely of '_'s. An operation symbol is classified as an infix when it both starts and ends with a '_', and as a prefix or postfix when it only ends, respectively starts, with a '_'. It is galled an 'outfix' when '_' only occurs internally.

There is one built-in constant symbol, 'nothing', and there are two built-in infix operation symbols, '_ | _', '_ & _'.

M.1.2 Variables

Variables are sequences of letters, here written in *this italic font*, optionally followed by primes $'$ and/or a numerical subscript or suffix.

M.1.3 Titles

Titles are sequences of words, here Capitalized and written in **This Bold Font**.

M.1.4 Marks

The marks used in the meta-notation consist of:

, . ; : :- :: () $= \leq \geq \rightarrow \Rightarrow \square$ / * *(continued)* **closed except open includes: introduces: needs: privately** *associative commu-*

tative disjoint for idempotent individual linear partial strict total unit is .

A pair of grouping parentheses ( ) may be replaced by a vertical rule to the left of the grouped material. Horizontal rules separate formal specification from informal comments. Reference numbers for parts of specifications have no formal significance.

M.2 Sentences

A sentence is essentially a Horn clause involving formulae that assert equality, sort inclusion, or individual inclusion between the values of terms. The variables occurring in the terms range over all values, not only over individuals. The universal quantification is left implicit.

M.2.1 Terms

Terms consist essentially of constant symbols, variables, and applications of operation symbols to subterms. We use 'mixfix' notation, writing the application of a operation symbol $S_0\_\ldots\_S_n$ to terms $T_1, \ldots, T_n$ as $S_0T_1\ldots T_nS_n$. Infixes have weaker precedence than prefixes, which themselves have weaker precedence than postfixes. Grouping parentheses ( ) may be inserted for further disambiguation. Parentheses may also be omitted when alternative ways of reinserting them lead to the same interpretation. E.g., the operation '_ | _' is associative, so we may write 'x | y | z' without disambiguating the grouping.

The value of a term is determined by the interpretation of the variables that occur in it. Such a value may be an individual (which is regarded as a special kind of sort), a vacuous sort, or a proper sort that includes some individuals.

The value of the constant 'nothing' is a vacuous sort, included in all other sorts. Operations map sorts to sorts, preserving sort inclusion. '_ | _' is sort union and '_ & _' is sort intersection; they are the join and meet, respectively, of the sort lattice, and enjoy the usual properties of set union and intersection.

M.2.2 Formulae

'$T_1 = T_2$' asserts that the values of the terms T_1 and T_2 are the same (individuals or sorts).

'$T_1 \leq T_2$' asserts that the value of the term T_1 is a subsort of that of the term T_2; so does '$T_2 \geq T_1$'. Sort inclusion is the partial order of the sort lattice.

'$T_1 : T_2$' asserts that the value of the term T_1 is an individual included in the (sort) value of the term T_2; so does 'T_1 :- T_2'. Thus '$T : T$' merely asserts that the value of T is an individual.

The mark '$\square$' (read as 'filled in later') in a term abbreviates the other side of the enclosing equation. Thus '$T_2 = T_1 \mid \square$' specifies the same as '$T_2 = T_1 \mid T_2$' (which is equivalent to '$T_2 \geq T_1$').

The mark '*disjoint*' following an equation or inclusion '$T = T_1 \mid \ldots \mid T_n$' abbreviates equations asserting vacuity of the pairwise intersections of the T_i. The mark '*individual*' abbreviates equations asserting that each T_i is an individual, as well as their disjointness.

'$F_1 \;; \ldots ; F_n$' is the conjunction of the formulae $F_1 \;; \ldots ; F_n$. Conjunctions with a common term may be abbreviated, e.g., '$x,\ y : z$' abbreviates '$x : z;\ y : z$' and '$x : y = z$' abbreviates '$x : y;\ y = z$'.

M.2.3 Clauses

A (generalized Horn) clause '$F_1; \ldots ; F_n \Rightarrow C_1; \ldots ; C_n$' asserts that whenever all the antecedent formulae F_i hold, so do all the consequent clauses (or formulae) C_j. Note that clauses cannot be nested to the left of $\Rightarrow$, so '$F_1 \Rightarrow F_2 \Rightarrow F_3$' is unambiguously grouped as '$F_1 \Rightarrow (F_2 \Rightarrow F_3)$'.

We restrict the interpretation of a variable V to individuals of some sort T in a clause C by specifying '$V : T \Rightarrow C$'. Alternatively we may simply replace some occurrence of V as an argument in C by '$V : T$'. We restrict V to subsorts of T by writing '$V < T$' instead of '$V : T$'.

M.2.4 Functionalities

A functionality clause '$S :: T_1, \ldots , T_n \rightarrow T$' specifies that the value of any application of S is included in T whenever the values of the argument terms are included in the T_i. It does *not* by itself indicate whether the value might be an individual, a proper sort, or a vacuous sort.

Such a functionality may be augmented by the following attributes:

***strict*:** the value is nothing when any argument is nothing;

***linear*:** the value on a union of two sorts is the union of the values on each sort separately, and similarly for intersections;

***total*:** the value is an individual when all arguments are individuals; moreover, S is *strict* and *linear*.

***partial*:** as for *total*, except that the value may also be a vacuous sort when the arguments are individuals.

When S is binary, we may use the following attributes (following OBJ3): *associative*, *commutative*, *idempotent*, and *unit is* T'. These attributes have a similar meaning when S is unary and the argument sort is a tuple sort, such as 'T^+' or '(T_1, T_2)'. (See the appendix on data notation for the notation for tuples, which is not regarded as a part of the meta-notation itself.)

In all cases, the attributes only apply when all arguments are included in the sorts specified in the functionality. For instance, consider:

```
product _ ::
  (number, number) → number (total, associative, commutative, unit is 1) ,
  (matrix, matrix) → matrix (partial, associative) .
```

which also illustrates how several functionalities for the same symbol can be specified together.

It is straightforward to translate ordinary many-sorted algebraic specifications into our meta-notation, using functionalities and attributes; similarly for order-sorted specifications [18] written in OBJ3 [20].

M.3 Specifications

A modular specification S is of the form '$B\ M_1 \ldots M_n$', where B is a basic specification, and the M_i are modules. Either B or the M_i (but not both) may be absent. B is inherited by all the M_i.

Each symbol stands for the same value or operation throughout a specification—except for symbols introduced 'privately'. All the symbols (but not the variables) used in a module have to be explicitly introduced: either in the module itself, or in an outer basic specification, or in a referenced module.

M.3.1 Basic Specifications

A basic specification B may introduce symbols, assert sentences, and impose (initial) constraints on subspecifications. The meta-notation for basic specifications is as follows.

'**introduces:** $S_1, \ldots, S_n$.' introduces the indicated symbols, which stand for constants and/or operations. Also '**privately introduces:** S_1, $\ldots$, S_n .' introduces the indicated symbols, but here the enclosing module translates them to 'new' symbols, so that they cannot clash with symbols specified in other modules.

'S . ' asserts the sentence S as an axiom, to hold for any assignment of values to the variables that occur in it. Omitting S gives the empty specification '.'.

'$B_1 \ldots B_n$' specifies all that the basic specifications $B_1, \ldots, B_n$ specify, i.e., it is their union. The order of the B_i is irrelevant, so symbols may be used before they are introduced.

'**includes:** $R_1, \ldots, R_n$.' specifies the same as all the modules indicated by the references R_i. '**needs:** $R_1, \ldots, R_n$.' is similar to '**introduces:** $R_1, \ldots, R_n$.', except that it is not transitive: symbols introduced in the modules referenced by the R_i are not regarded as being automatically available for use in modules that reference the enclosing module.

'**grammar:** S' augments the basic specification S with standard specifications of strings and trees from data notation, and with the introduction of each constant symbol that occurs as the left-hand-side of an equation in S. Similarly when S is a series of modules.

'**closed** .' specifies the constraint that the enclosing module is to have a 'standard' (i.e., initial) interpretation. This means that it must be possible, using the specified symbols, to express every *individual* that is included in some expressible sort ('no junk'), and moreover that terms have equal/included/individual values only when that logically follows from the specified axioms ('no confusion'). '**closed except** $R_1, \ldots, R_n$.' specifies a similar constraint, but leaves the (sub)modules referenced by the R_i open, so that they may be specialized in extensions of the specification. '**open** .' merely indicates the the enclosing module is not to be closed.

M.3.2 Modules

A module M is of the form 'T S', where T is a title (or a series of titles, separated by '/') that identifies the specification S.

Modules may be specified incrementally, in any order. To show that a module is continuing an earlier specification with the same identification, the mark '*(continued)*' is appended to its title.

Modules may also be nested, in which case an inner module inherits the basic specifications of all the enclosing modules, and the series of titles that identifies the immediately enclosing module. Titles are not inherited when T starts with a '/'. A part of a title starting with '/∗/' is ignored, it merely indicates the form of the titles of submodules.

Parameterization of modules is rather implicit: unconstrained submodules, specified as '**open** .', can always be specialized.

A series of titles '$T_1/\ldots/T_n$ refers to a module (together with all its submodules). A common prefix of the titles of the enclosing module and of the referenced module may be omitted. In particular, 'sibling' modules in a nest can be referenced using single titles. '$T/(T_1,\ldots,T_n)$' refers to the collection of modules '$T/T_1,\ldots,T/T_n$'.

'T (S'_1 *for* $S_1,\ldots,S'_n$ *for* S_n)' refers to the same module as the title(s) T, but with all the symbols S_i translated to S'_i. Each S_i must be specified by the module referenced by T. Identity translations 'S_i *for* S_i' may be abbreviated to S_i, as in 'T ($S_1,\ldots,S_n$)' which indicates that the module referenced by T specifies at least all the symbols $S_1,\ldots,S_n$.

Appendix N

Summary of Variable Restrictions

In appendices A-F, we have consistently restricted the interpretation of some variables to individuals of a particular sort. The following is a list of such restrictions; the indicated sorts are exclusively ranged over by the shown variables.

G	:	global-register
R, R', R'', RI'''	:	register
T	:	Type
a, a', a_n, a'_n, a''_n, a'''_n a_e, a'_e, a''_e, a'''_e, r, r', r''	:	general-register
b, b', b''	:	bindings
c, c', c''	:	commitment
ce	:	cell
ct	:	truth-value-cell
ci	:	integer-cell
cz	:	was-zero
cn	:	was-negative
d, d'	:	symbol-table
e, e', e'', e'''	:	block
f, f'	:	frozen

g	:	globals
h, h', h_n, h'_n, h''_n, h'''_n, h_e, h'_e, $h''{}_e$, h'''_e	:	data-type
i, i', i''	:	integer
il, ol	:	[integer] list
io, io', io''	:	input-output
j, j', n, n', n'', nw	:	natural
k, k'	:	token
l, l', l'', l'''	:	linenumber
l_n	:	linenumber-complete
l_e	:	linenumber-escape
l_f	:	linenumber-fail
l_u	:	linenumber-unfold
lt	:	linenumber*
m, m'	:	storage-map
m_a	:	state
m_p, m'_p, m''_p	:	spare-state
nt, nt'	:	natural*
p, p', p'', prg	:	program
pc	:	program-counter
q, q', q''	:	memory
s, s', s''	:	storage
se	:	space
t, t', t''	:	data
u, u_n, u_e, u_f	:	cleanup
v, v'	:	datum
w, w', w''	:	windows
z_n, z'_n, z''_n, z'''_n, z_e, z'_e, z''_e, z'''_e	:	truth-value

Index

( ), 260
(_ , _), 260
0, 254
Act, 61
Alternatives, 221
a _, 245
a-consistent _ _ _ _ , 131
a-count _ _ _, 93
a-post-condition _ _ _ _ _ _ _ _ _ _ _ , 135
a-size, 88
a-state _ _ _, 87
a-state, 87
abstract _ _ _ _ _ _ _, 195
abstraction of _, 251
abstraction-free _ , 85
abstraction-type _ _ _ _ _ _, 84
abstraction-type, 84
abstraction, 64, 251
ac-less _ _ _ _ _ _ _ _ _ _, 133
ac-state _ _ _ _ _ _, 87
ac-state, 87
activate _, 199
actualize-formal _ ,, 223
actualize-formal-in _, 223
actualize-formals _, 223
actualize-formals-in _, 223
act, 241
all _, 254
allocate _, 252
allocate-for _ , 223
allocate-for-primitive _, 223
an _, 245
any _, 254
application _ to _, 250
argument, 77
arg, 77
at-code _ _ _ _ _, 88
Binary-Operator, 221
Binary, 62
Block
  HypoPL, 198
  Mini-Ada, 221
b-abs _ _ _ _ , 126
basic-post-condition _ _ _ _ _ _ _ _ _ , 135
basic-pre-condition _ _ _ _ _ _ _ _ _ , 135
bc-abs _, 133
binary-code _ _ _ _ _ _, 121
binary-count _ _ _, 102
binary-operation _ _ _, 74
bind _ to _, 247
bindable, 248
bindings, 64, 248
block-find-count _ _, 101
block, 87
both _, 254
branchequal _, 77
branchlessthan _, 77

Control-Operator, 221
c-abs _ , 126
c-size, 88
call-sequence _ _ _ _ _ _ _ _ _, 92
call, 77
cc-less _ _ _ _ _, 133
cef, 77
cell-type, 84
cell, 63, 249
character, 255
check _, 246
cleaned-up _ _ _ _ _ _ , 135
cleanup, 87
closure _, 250
closure-abstraction _ _ _, 64
code-size _, 87
coercively _, 199
combine _ _ _ _, 90
combinecommit _, 90
commitment _, 64
commitment, 62
commits, 77
committend, 62
commit, 242
compare _ with _, 77
compare-blocks _ _ _ _, 87
compare-data-types _ _ _ _, 85
compile _, 195
completed _ _ _ _ _ , 64
completed, 64
complete, 242
component#_ _, 260
concatenation _, 258
concatenation-code _ _ _ _, 88
count _, 260
cp, 77
Data, 62
Datum, 62
Declarations, 220
Declaration, 198
Dependent, 61
d-consistent _ _ _ _ , 131
d-count _ _ _, 98
d-post-condition _ _ _ _ _, 135
data-type _, 103
data-type , 85
data_ , 75
data, 64, 245, 246
datum _, 75
datum-type _, 103
datum, 64, 244
dependent _, 244
difference _ _ , 77
difference _, 255, 259
diverge, 242
Expressions, 221
Expression
    HypoPL, 198
    Mini-Ada, 221
e-abs _ _ _ _ , 126
e-size, 88
ec-abs _, 133
ec-less _ _ _ _ _ , 133
either _, 254
elaborate _, 223
elements _, 258
empty-list-code _ _, 88
empty-list, 258
empty-map, 259
empty-set, 259
empty-storage, 63
enact _, 250
entry _ _, 87
entry, 87
error, 87
err, 222

escape-reason, 222
escaped _ _ _ _, 64
escaped, 64
escape, 242
establish _, 199
evaluate _ _ _ _ _ _ _ _ , 118
evaluate _
  HypoPL, 199
  Mini-Ada, 223
evaluate-argument _ _ _ , 82
evaluate-movable _ _ _, 82
evaluated _ _ _ _, 73
execute _
  HypoPL, 199
  Mini-Ada, 223
execute-block _, 223
exhaust _, 223
exit, 222
Formal-In, 220
Formals-In, 220
Formals, 220
Formal, 220
f-post-condition _ _ _ _, 135
failed _ _ _, 64
failed, 64
fail, 242
false, 253
fetch _ _ _ , 82
final _ _ _ _ _, 65
finalize _ _ _ _ , 91
find _ _ _ , 120
find-consistent _ _ _ , 131
find-count _ _, 101
finished _ _ _, 78
firstfree, 77
free-register _, 86
frozen, 86
function-return-of _, 222
function-return, 222
furthermore _, 247
general-register, 77
give _, 245
given _ # _, 246
global-register, 77
globals _, 78
globals, 78
global, 77
h-size, 88
head _, 258
head-code _ _ _ _, 88
heap, 77
hp, 77
Identifier
  HypoPL, 199
  Mini-Ada, 222
Infix, 62
Integer
  HypoPL, 199
  Mini-Ada, 221
i-abs _ _ , 126
id _
  HypoPL, 199
  Mini-Ada, 223
if _ then _ else _, 254
indivisibly _, 243
input-output _, 64
input-output, 64
input, 77
insert _ _ , 87
instruction, 77
integer-cell _, 63
integer-cell-type, 84
integer-cell, 63
integer-type, 84
integer-value _
  HypoPL, 199

Mini-Ada, 223
integer, 255
intersection _, 259
investigate _, 223
items _, 258
item
HypoPL, 199
Mini-Ada, 222
it, 246
jump _, 77
linenumber-complete, 87
linenumber-escape, 87
linenumber-fail, 87
linenumber-unfold, 87
linenumber, 78
list of _, 257
list-type _, 84
list-type, 84
list, 257
load _ _ _ into _, 77
loadregisters, 77
m-abs _ _ _ _ _ _ _ _ _ _ _ _ , 126
m-earlier _ _, 135
map of _ to _, 259
map of _, 259
mapped-set _, 259
map, 259
maximum of _, 86
mc-abs _, 133
memory _, 78
memory, 78
minimum of _, 86
movable, 77
move _ to _, 77
mq-earlier _ _, 135
multi-evaluate _, 223
multi-evaluated _ _ _ _, 72
multi-investigate _ , 223
Names, 221
Name, 221
Nominator, 220
natural, 254
negation _, 255
next _ _ , 80
non-abstraction, 222
nonzero-integer, 255
not _, 254
Operation, 199
o-abs _ _ , 126
of _, 245
op-post-condition _ _ _ _ _, 135
operation-consistent _ _ _ , 131
operation-result _, 199
out-of-bound _, 78
output, 77
overlay _, 260
Primitive, 220
Program
HypoPL, 198
Mini-Ada, 220
page-id, 77
page, 78
parameter-less-closure _, 222
parameter-less-function, 222
parameter-less-procedure, 222
parameterized-function-closure _, 223
parameterized-function, 222
parameterized-procedure-closure _, 223
parameterized-procedure, 222
perdecessor _, 255
perform _ _ _ _ _ _ _ _ _ _ _ _, 104
post-condition _ _ _ _ , 135
pre-condition _ _ _ _ _ _ _ _ _ _ _ , 135
procedure-return, 222
program _, 78
program-counter _, 78

program-counter, 78
program, 78
putcommit _ _, 90
q-earlier _ _ _ _ , 135
range, 259
reg _, 77
registers up to _, 86
registers, 78
register, 77
regive, 245
result _ _ _ _ _ _ _ _ , 195
result, 195
return-address, 77
return-sequence _ _ _ , 92
returned-value-of _, 222
return, 77
run _ _ , 195
run _
    HypoPL, 199
    Mini-Ada, 223
Statements, 220
Statement, 198
s-abs _ _ , 126
s-size, 88
set of _, 258
set, 258
single-list-code _ _ _, 88
skip, 77
small-datum _, 75
sparc-final _ _ _ _, 80
sparc-run _ _ _, 195
sparc-state, 78
sp, 77
stack, 77
state, 64
staticlink, 77
step _ , 80
storable-abs _ , 126
storable-type _, 84
storable, 63, 249
storage _, 64
storage-map, 63
storage, 63, 249
store _ in _ _ _ , 77
store-abs _ _ , 126
storeable in _, 63
storeregisters, 77
store, 77
store _ in _, 249
string, 256
successor _, 86, 254
sum _ _, 77
sum _, 254, 255
symbol-table, 87
syntax-free, 257
Tuple, 61
Type, 62
t-abs _ _ _ _ , 126
t-size, 88
tail _, 258
tail-code _ _ _ _, 88
tc-abs _, 133
the _ bound to _, 248
the _ stored in _, 249
the _, 245
the-binary-operation-result-of _, 223
the-control-operation-completion-of _, 223
them, 246
there-is-given-a-procedure-return, 222
there-is-given-a-return, 222
there-is-given-an-exit, 222
token, 248
tree, 256
true, 253
truth-value-cell _, 63

truth-value-cell-type, 84
truth-value-cell, 63
truth-value-type, 84
truth-value, 253
tuple, 260
type _, 103
type, 84
Unary, 62
Unf, 61
u-count _ _ _ _ _ _ _ _ , 96
u-post-condition _ _ _ _ _ _ _ _ _ _ _ , 135
unary-code _ _ _ _ _ , 121
unary-count _ _, 102
unary-operation _ _, 74
unchanged _ _ _ _ _ , 135
uncommitted, 62
unf _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ , 113
unf-final _ _ _ _ _ _, 70
unfolding _, 242
unfold, 242
uninitialized, 63, 249
union _, 259
up-to _, 135
update _ _, 78
v-abs _ _ _ _ , 126
vc-abs _, 133
w-consistent _ _ _ _ , 131
w-count _ _ _, 97
was-negative, 78
was-zero, 78
wc-state _ _, 87
wc-state, 87
well-placed _ _ _, 131
windows _, 78
windows, 78
with _ _ _ _ _ _ _ , 117
[_ to _] _, 260
[_] _, 258
[perhaps using _] act, 241
_ and then _, 244
_ and _, 243
_ at _ default _, 78
_ at _, 260
_ before _, 248
_ hence _, 247
_ is greater than _, 253
_ is in _, 78, 259
_ is included in _, 259
_ is less than _, 86, 253, 255
_ is submap of _ , 123
_ is _, 123, 253–255, 259
_ leq _ , 135
_ omitting _, 259, 260
_ or _, 243
_ restricted to _, 259, 260
_ then _, 246
_ thence _, 251
_ trap _, 244

Bibliography

[1] Harald Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Eighteenth Symposium on Principles of Programming Languages*, pages 104–118. ACM Press, January 1991. To appear in ACM TOPLAS, Transactions on Programming Languages and Systems.

[3] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[4] Rudolf Berghammer, Herbert Ehler, and Hans Zierer. Towards an algebraic specification of code generation. *Science of Computer Programming*, 11:45–63, 1988.

[5] William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, 1989.

[6] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice-Hall, 1982.

[7] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proc. ESOP'90, European Symposium on Programming*, pages 70–87. Springer-Verlag (*LNCS* 432), 1990.

[8] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] Rod M. Burstall and Peter J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence, Vol. 4*, pages 17–43. Edinburgh University Press, 1969.

[10] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation.* To appear. A preliminary version of this paper was presented at OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, New Orleans, Louisiana, October, 1989.

[11] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[12] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[13] Mads Dam and Frank Jensen. Compiler generation from relational semantics. In *Proc. ESOP'86, European Symposium on Programming*, pages 1–29. Springer-Verlag (*LNCS* 213), 1986.

[14] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS'86, First Symposium on Logic in Computer Science*, pages 193–205, June 1986.

[15] Jean D. Ichbiah et al. *Reference Manual for the Ada Programming Language.* US DoD, July 1982.

[16] Susan Even and David A. Schmidt. Type inference for action semantics. In *Proc, ESOP'90, European Symposium on Programming*, pages 118–133. Springer-Verlag (*LNCS* 432), 1990.

[17] Anders Gammelgaard and Flemming Nielson. Verification of the level 0 compiling specification. Technical report, Department of Computer Science, Aarhus University, July 1990.

[18] Joseph A. Goguen and José Meseguer. Order-sorted algebra: Algebraic theory of polymorphism. *Journal of Symbolic Logic*, 51:844–845, 1986. Abstract.

[19] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology Volume IV*, pages 80–149. Prentice-Hall, 1978.

[20] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Lab., SRI International, 1988.

[21] Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

[22] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[23] Danny Goodman. *The Complete HyperCard Handbook.* Bantam, 1987.

[24] John Hannan. Making abstract machines less abstract. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 618–635. Springer-Verlag (*LNCS* 523), 1991.

[25] John Hannan. Staging transformations for abstract machines. In *Proc, ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 130–141. Sigplan Notices, 1991.

[26] John Hannan and Dale Miller. From operational semantics to abtract machines, *Journal of Mathmatical Structures in Computer Science*, To appear, 1991.

[27] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[28] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[29] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In *Proc. Hardware Specification, Verification and Synthesis: Mathmatical Aspects*, pages 177–201, July 1989.

[30] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical report, University of Cambridge, Computer Laboratory, England, March 1989.

[31] Gilles Kahn. Natural semantics. In *Proc. STACS'87*, pages 22–39. Springer-Verlag (*LNCS* 247), 1987.

[32] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Sixteenth Symposium on Principles of Programming Languages*, pages 281–292. ACM Press, January 1989.

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[34] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[35] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[36] Peter Lee and Uwe F. Pleban. A realistic compiler generator based on high-level semantics. In *Fourteenth Symposium on Principles of Programming Languages*, pages 284–295. ACM Press, January 1987.

[37] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proc. Symposium in Applied mathematics of the American Mathmatical Society*, pages 33–41, April 1966.

[38] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[39] Sun Microsystems. A RISC tutorial. Technical Report 800-1795-10, revision A, May 1988.

[40] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[41] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[42] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[43] Francis Lockwood Morris. Advice on structuring compilers and proving them correct. In *Symposium on Principles of Programming Languages*, pages 144–152. ACM Press, October 1973.

[44] Peter D. Mosses. SIS—semantics implementation system. Technical Report Daimi MD–30, Computer Science Department, Aarhus University, 1979.

[45] Peter D. Mosses. A constructive approach to compiler correctness. In *Proc. Seventh Colloquium of Automuta, Languages, and Programming*, pages 449–469, July 1980.

[46] Peter D. Mosses. Abstract semantic algebras! In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts II (Garmisch-Partenkirchen, l982)*, pages 45–70. North-Holland, 1983.

[47] Peter D. Mosses. A basic abstract semantic algebra. In *Proc. Int. Symp. on Semantics of Data Types (Sophia-Antipolis)*, pages 87–107. Springer-Verlag (*LNCS* 173), 1984.

[48] Peter D. Mosses. Unified algebras and action semantics. In *Proc STACS'89*, pages 17–35. Springer-Verlag (*LNCS* 349), 1989.

[49] Peter D. Mosses. Unified algebras and institutions. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 304–312, 1989.

[50] Peter D. Mosses. Unified algebras and modules. In *Sixteenth Symposium on Principles of Programming Languages*, pages 329–343. ACM Press, January 1989.

[51] Peter D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–631. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[52] Peter D. Mosses. Action semantics. Technical Report DAIMI FN-48, Computer Science Department, Aarhus University, November 1991.

[53] Peter D. Mosses. An introduction to action semantics. Technical Report DAIMI PB-370, Computer Science Department, Aarhus University, 1991. Lecture Notes for the Marktoberdorf'91 Summer School, to be published in the Proceedings of the Summer School by Springer-Verlag (Series F).

[54] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992. *Tracts in Theoretical Computer Science.*

[55] Peter D. Mosses and David A. Watt. The use of action semantics. In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts III (Gl. Avernæs 1986)*, pages 135–163. North-Holland, 1987.

[56] Flemming Nielson and Hanne Riis Nielson. The TML-approach to compiler-compilers. Technical report, The Technical University of Denmark, November 1988.

[57] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[58] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages.* Cambridge University Press, 1992.

[59] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[60] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A Formal Introduction.* Wiley & Sons, 1992.

[61] Hewlett Packard. Precision architecture and instruction. Technical Report 09740-90014, June 1987.

[62] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proc. ICCL'92, Fourth IEEE International Conference on Computer Languages*, pages 117–126, Oakland, California, April 1992.

[63] Jens Palsberg. A provably correct compiler generator. In *Proc. ESOP'92, European Symposium on Programming*, pages 418–434. Springer-Verlag (*LNCS* 582), Rennes, France, February 1992.

[64] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990.

[65] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[66] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, pages 325–341. Springer-Verlag (*LNCS* 512), Geneva, Switzerland, July 1991.

[67] Jens Palsberg and Michael I. Schwartzbach. Three discussions on object-oriented typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992.

[68] Lawrence Paulson. A semantics-directed compiler generator. In *Ninth Symposium on Principles of Programming Languages*, pages 224–233. ACM Press, January 1982.

[69] Uwe F. Pleban. Compiler prototyping using formal semantics. In *Proc, ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 94–105. Sigplan Notices, 1984.

[70] Uwe F. Pleban and Peter Lee. On the use of LISP in implementing denotational semantics. In *Proc. ACM Conference on LISP and Functional Programming*, pages 233–248, August 1986.

[71] Uwe F. Pleban and Peter Lee. High-level semantics, an integrated approach to programming language semantics and the specification of implementations. In *Proc. Mathmatical Foundations of Programming Language Semantics*, pages 550–571, April 1987.

[72] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proc, SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 222–232, June 1988.

[73] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[74] Wolfgang Polak. *Compiler Specification and Verification.* Springer-Verlag (*LNCS* 213), 1981.

[75] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[76] David A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[77] David A. Schmidt. Detecting stack-based environments in denotational semantics. *Science of Computer Programming*, 11:107–131, 1988.

[78] Uwe Schmidt and Reinhard Völler. A multi-language compiler system with automatically generated codegenerators. In *Proc. ACM SIGPLAN'84 Symposium on Compiler Construction.* Sigplan Notices, 1984.

[79] Uwe Schmidt and Reinhard Völler. Experience with VDM in Norsk Data. In *VDM'87. VDM—A Formal Method at Work*, pages 49–62. Springer-Verlag (*LNCS* 252), March 1987.

[80] Gert Smolka. Order-sorted horn logic semantics and deduction. FB Informatik, Universität Kaiserslautern, September 1986.

[81] William Stallings. *Reduced Instruction Set Computers.* IEEE Computer Society Press, 1986.

[82] Joseph E. Stoy. *Denotational Semantics: The ScottStrachey Approach to Programming Language Theory.* MIT Press, 1977.

[83] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[84] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.

[85] Mads Tofte. *Compiler Generators.* Springer-Verlag, 1990.

[86] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag (*LNCS* 201), 1985.

[87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, pages 227–241, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Acadamic Publishers, June, 1991.

[88] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly, 1991.

[89] Mitchell Wand. A semantic prototyping system. In *Proc. ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 213–221. Sigplan Notices, 1984.

[90] David Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.

[91] Pierre Weis. *Le système SAM, Métacompilation très efficace á l'aide d'Opéateurs Sémantiques.* PhD thesis, Universite Paris VII, November 1987.

[92] Niclaus Wirth. The programming language Pascal. *Acta Informatica*, 1:33–63, 1971.

[93] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

[94] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, New York, 1985.

[95] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.