Aspects of Programming

Even-André Karlsson
Division of Computer Science
Norwegian Institute of Technology

August 10, 1990
Abstract

There has always been a great interest in programming languages and support environments. For programming languages it suffices to mention the active research in functional, logical and object-oriented paradigms. For support environments the research can be broadly categorized into two different approaches. One is language based environments which are tightly integrated to support one specific language. The other is the toolbox approach which can support different languages.

One requirement to all languages and support environment is that they not only support programming-in-the-small, but also provide adequate support for programming-in-the-large and system maintenance.

This work is an attempt to orthogonally apply the constructs found most fundamental in the programming-in-the-small area in a new language, Nel. The emphasis is put on expressiveness, toning down efficiency. These constructs are then compared with the requirements from the other two domains, and to a large extend found to satisfy these requirements.

A language based programming support environment including automatic support for change-oriented-versioning is also specified for Nel.
Contents

1 Introduction

1.1 The software development scenario ........................................... 1
1.2 Intention .............................................................................. 2
  1.2.1 What is new? .............................................................. 3
  1.2.2 Disclaimer ................................................................. 3
  1.2.3 The rest of this thesis ................................................... 4
1.3 Requirements ....................................................................... 5
  1.3.1 One language and one environment ................................. 5
  1.3.2 Computational model ................................................... 6
  1.3.3 Multi User support ....................................................... 6
  1.3.4 Programming styles and paradigms ............................... 6
  1.3.5 User interaction .......................................................... 7
  1.3.6 Persistence ................................................................... 7
  1.3.7 Versioning .................................................................... 7
  1.3.8 Naming ........................................................................ 8
  1.3.9 Typing ........................................................................ 8
  1.3.10 Information hiding ...................................................... 9
1.4 Some similar systems and related work .................................... 9
  1.4.1 Unifying approaches ..................................................... 9
  1.4.2 Ada ............................................................................ 10
  1.4.3 Cedar ......................................................................... 10
  1.4.4 Related work ............................................................. 11

2 The Nel language ...................................................................... 13
3.6 Recursive values ............................................. 47
3.7 Environments and object oriented languages ............... 48
3.8 Unification .................................................. 50
3.9 Concurrent constructs ....................................... 53
3.10 Software development in NEL ................................ 55

4 The NEL Programming support environment .......................... 59
4.1 Introduction .................................................. 59
4.2 Controlling DEI ............................................. 60
4.3 Interpreting in DEI ......................................... 61
4.4 Visiting a function .......................................... 63
4.5 Tracing in DEI ............................................... 64
4.6 Versioning in DEI ........................................... 66
4.7 Other values in DEI ......................................... 72
4.8 Multi user control .......................................... 74

5 Semantics ................................................................ 81
5.1 Introduction .................................................. 81
5.2 Sequential semantics .......................................... 82
   5.2.1 Notation ................................................ 82
   5.2.2 Sequential semantic domains ......................... 83
   5.2.3 The semantic function E ............................... 84
   5.2.4 Recursive values ....................................... 84
   5.2.5 Auxilliary functions .................................... 86
   5.2.6 Exceptions ............................................... 87
   5.2.7 Type checking functions ............................... 88
   5.2.8 Simple values and their operations ................... 88
   5.2.9 Declarations ........................................... 90
   5.2.10 Constructed values ..................................... 91
   5.2.11 Access control ......................................... 92
   5.2.12 Equality, subset and assignment ..................... 93
   5.2.13 Copy, value and type .................................. 97
5.2.14 Union and unification ........................................... 99
5.2.15 Sequential control flow ........................................ 102
5.2.16 Functions and continuations .................................... 103
5.2.17 Input and output ................................................. 104
5.3 Parallelism ............................................................. 106
5.3.1 The semantic domains ............................................. 106
5.3.2 The interleaving model ............................................ 107
5.3.3 Changes to the Sequential model .................................. 108
5.3.4 The parallel constructs .......................................... 109
5.4 Explicit continuations ............................................... 116
5.4.1 The semantic domains ............................................. 117
5.4.2 The semantic function ........................................... 117
5.4.3 Changes to the semantic equations .............................. 117
5.4.4 Semantic equations ................................................ 118
5.4.5 Storing the run time stacks ...................................... 119

6 Conclusion .................................................................. 120
6.1 Efficiency ............................................................... 120
6.1.1 The results of partial evaluation ................................ 122
6.1.2 Abstract interpretation .......................................... 123
6.2 Niel as an intermediate language ................................ 125
6.3 Implementation and use ............................................. 126
6.4 Assessment ............................................................. 126

A The complete syntax .................................................. 128

B What was left out ....................................................... 129
B.1 Real time and priority ............................................. 129
B.2 Transaction facilities .............................................. 129
B.3 Deep assignment ..................................................... 130
B.4 Universal identifiers ................................................. 130
B.5 Widening ............................................................... 131
B.6 Non-determinism ....................................................... 131
C  Change oriented versioning  133

D  Types  136
   D.1  Non functional types  137
   D.2  Function types  141

E  Nel and Prolog  144
Acknowledgment

I'd like to thank my advisor, professor Reidar Conradi, for inviting me to a dr.ing. study, for believing in the work, and for stimulating discussions in particular his emphasis on programming-in-the-large and system maintenance. I will also thank him for all the challenging work assignments he has provided, in particular the possibility to teach the courses “Programming Languages” and “Operating Systems”. This has given me invaluable experience. The beginning to many of the ideas in this work were established through the lecturing.

This work has been done in the context of the EPOS project, and the members of the project group have been invaluable discussion partners. I'd like to thank them all in general. I would in particular like to thank Tor M. Didriksen for his early interest in the Nel language, and Anund Lie and Bjørn Gulla for many stimulating discussions on change oriented versioning. A special thanks goes to Per Holager, the originator of the change oriented versioning, the idea which has made the EPOS project a very interesting project to be affiliated with.

I'd also like to thank Svein Erik Bratsberg and Erik Odberg for implementing the first version of Nel, and providing good support in my teaching of Programming Languages. The students enduring my far from perfect teaching also deserve to be mentioned.

During the period of my dr.ing. study, I have mainly been supported by NTH as a scientific assistant for one year, and by a NTNF fellowship for the last two.
Chapter 1

Introduction

1.1 The software development scenario

The development of programming languages and paradigms, and their supporting tools and environments has, from the time of the first electronic computers, been an important part of computer science.

The programming language is the tool with which software products are built. The research to find the most appropriate paradigms and language constructs is one of the core fields of computer science. An example of this interest is reflected in the ACM Computing Surveys Special Issue on Programming Language Paradigms from September 1989. The aspect of expressing single algorithms can be captured under the umbrella Programming-in-the-small (PITS) [DEREMER76].

Language design is a trade off between compiler technology and efficient object code on one side and expressive power and orthogonal constructs on the other. This has often restricted the expressive power of the languages and introduced specialized constructs. This again has created the need for language extensions in the form of e.g. preprocessors. Recently the interest in languages which focus more on expressive power and orthogonality has increased. In connection with this interest, the search for heuristic techniques for generating efficient code for these languages has evolved. The primary example is functional languages and abstract interpretation.

The role of the other tools in the programming environment is to help the programmer to use the programming language. Tools should be simple to use, and there should not be too many of them. When the number and complexity of the tools increase, managing which tool to use, how and when to use it, becomes a problem in itself. The support environment for the programming language should facilitate the task of programming and help the programmer to use the language and tools correctly. The support environments can be divided into two main categories;

- Language dependent, i.e. the environment is designed to support a specific programming language, and can thus be better integrated with this language. A typical example is the Smalltalk environment [GOLDBERG83].
• Language independent, i.e. the environment does not have a coupling to any specific language, but is supposed to support every language equally well. Unix is an example of this category.

As a program grows larger there is a need to split it into components which are later reassembled to the complete program. This description of the entire software product as a collection of smaller components, and how they are related can be termed Programming-in-the-large (PITL) [DEREMER76]. There has also been interest in finding appropriate language constructs for describing the components and their interconnection, i.e. module interconnection languages [PRIETO-DIAZ86].

As software products become larger and older, the problem of managing these products with all their changes and additions has increased. This field could be called Software Maintenance (SM). There has been a growing interest over the last two decades to extend the programming support environments to support also this phase of software development. As the maintenance can easily represent 60 to 80% of the total cost and time of large software products, it is important to have appropriate tools for this phase.

Up till recently there has been few unifying approaches to the problems in the areas of PITS, PITL and SM. These three aspect of software construction have been treated rather separately, with different languages and different tools.

1.2 Intention

The intention of this work is to investigate the possibility of designing a programming language, Nel, which is flexible enough to cover some aspects of PITL and SM beside PITS.

Nel does not extend existing languages in any fundamentally new dimension, but all constructs incorporated are orthogonal. This is used to demonstrate that most PITL and SM concepts can be integrated into a PITS language and environment by using orthogonal language constructs.

The disadvantage is that there is no guarantee that software products written in this system will eventually be transformed to efficient code. But heuristic techniques based on semantic definitions (abstract interpretation and partial evaluation) could in most cases give equally good results, and for the remaining cases the system should be able to give the user a warning.

To start with a PITS language is a natural approach, as PITS languages and their tools are the richest, most developed, and represents a necessity in any complete system. Furthermore the development of PITS languages is by far the richest and best understood.

The aim is also to get a better understanding of basic principles of programming languages, their definition, use and tools. By “use” is understood both PITS, and the connection with PITL and SM. In the process of discussing different PITS constructs, we also get a better foundation for finding the right constructs which
will also be applicable for PITL and SM.

By making Nel orthogonal and expressive, we should be able to relate it to the other languages, and use it as an intermediate language for a programming environment generator. The partial evaluation techniques should be capable of giving the generated system an acceptable performance. We should also be able to reuse the Nel tools, or extend them to tool generators.

There is of course more to a software product than the program text. Requirements, design, technical and user documentation, and change information are some of the other crucial objects, and they are of course equally important to manage properly. These objects are however outside the scope of this thesis, as they mostly do not have a formally specified syntax and meaning which are well enough defined to be processed automatically. The maintenance of these objects are more in the scope of language independent environments.

1.2.1 What is new?

Every dr. ing. thesis is supposed to have some aspect of innovation, or originality. The research of programming languages and their support environment is quite established, and to claim that there is anything fundamentally new in this thesis would be an exaggeration. Some points might be of some novelty though. Of these I will emphasize the following:

- The uniform treatment of values collected in a value lattice, and its used to both represent type and value information about variables.
- The attempt to integrate PITL and SM into a PITS language.
- The application of change oriented versioning to program text, and thus the integration of versioning into the programming language.
- Providing for multiple users in a tightly integrated environment by concurrent processes in the language.

The significance of each of these points is discussed in greater depth in their right context.

1.2.2 Disclaimer

The title of this work, "Aspects of programming", is intended to indicate that it is a general discussion of different programming language concepts and paradigms. Aspects are here to be taken literally, i.e. only some features of some of the different paradigms are discussed. It is impossible to cover everything, as this work actually relates to too many aspects of programming. The work is also rather practical, as most of the work is done on the syntactic level, and concerned with expressiveness.
Some, maybe most, aspects are treated quite, perhaps intolerably, superficially. The justification for this is that a more profound treatment of each of the aspect that I touch would require a text many times the size of this, which is beyond my capacity regarding both time and knowledge. The aspects touched are those that I found most relevant to put this work into perspective.

1.2.3 The rest of this thesis

The rest of this introduction contains some requirements for a language combining PITS with PITL and SM, and a discussion of how these requirements are covered by conventional programming paradigms and support environments.

Chapter 2 gives an informal introduction to the Nel language. Nel serves as an example language for the rest of the thesis. Nel supports different programming paradigms, PITL and SM by providing mechanisms instead of policies, and stressing the orthogonality of the constructs.

Chapter 3 contains examples of the use of Nel, both for PITS, PITL and SM. It also relates the constructs in Nel to those found in other programming languages and environments.

Chapter 4 describes the programming in the large support environment for Nel. There is as little difference as possible between the level of the programming language and the program support environment. In this chapter the change oriented versioning model is used on programs.

In chapter 5 the semantic definition of Nel is given in a denotational style. This contains a considerable amount of detail, and some of the algorithms are quite complex.

The conclusions of the thesis is drawn in chapter 6. Here also some indication of further work is outlined.

The appendices contains a complete syntax of the Nel language, an overview and short discussion of constructs not included in Nel, an overview of change oriented versioning, an overview of some work on types in programming languages, and finally a mapping of Prolog to Nel.

In the rest of this work, some languages and systems will be repeatedly referred, and it is chosen to include the standard references to these systems here. These are Ada\(^1\) [ANSI/MIL83], Algol68 [WIJNGAARDEN75], C++ [STROUSTRUP86], Eiffel [MEYER88], Emacs [STALLMAN81], Lisp [MCCARTHY60,63] and its environments [TEITELMAN81], Make [FELDMAN79], Miranda\(^2\) [TURNER86], ML [MILNER84], Pascal [JENSEN74], Prolog [COLMERAUER70,73], RCS [TICHY-82], Russell [DEMERS83], SCCS [ROCHKIND75], Scheme [REES86], Simula [DAHL70], Smalltalk [GOLDBERG83], and Unix\(^3\) [RITCHIE74].

\(^1\)Ada is a trademark of the US Government
\(^2\)Miranda is a trademark of Research Software Limited
\(^3\)Unix is a trademark of AT&T
1.3 Requirements

This section contains a discussion of some requirements for a language combining PITS with PITL and SM. It also relates these requirements to well known existing systems.

1.3.1 One language and one environment

The idea is to have one language and one environment with which the user interacts. Unix is a prime example of the multitude of sub-languages that a user in a language independent environment has to relate to:

- RCS/SCCS, which is used to store distinct, mostly sequential revisions of the source, and parameters must be given to select the correct revision.

- *Make* records the interconnection between the components of the system, and is used to rebuild the system. *Make* resembles a function which is invoked to reproduce the system.

- Preprocessors (conditional compilation) are usually used to record mostly orthogonal variation in the source, e.g. different operating systems.

- C++ could be the source language in which the "real" program is written.

- Shells provide the command language and interpreter for interactively manipulating software objects in the programming-in-the-large domain. Shells usually also contain small programming languages to make batch files.

- Debuggers are usually equipped with their own set of data manipulation primitives. Some are even so advanced as to include an appropriate language in which repeated commands can be written.

- Editors contain powerful operations to manipulate text, which also are provided by to a much lesser extent by debuggers and shells. Furthermore any editor has a macro facility to write small programs.

Examples of language based environments are Smalltalk and Lisp environments. Here the user can relate to one language, and usually one set of editing commands. The disadvantages with these two systems are their lack of multi user and software maintenance support.

Thus, we believe that the programming language should be the same as the command language, and the data objects in the program should be the same as those in the programming environment.
1.3.2 Computational model

The computational model should allow multiple users / processes. This is important if PITL and SM are to be supported, as all larger program systems are constructed in cooperation by more persons. The processes should cooperate on a "global" state, i.e. the software product.

The lack of multi user support is one of the disadvantages of Lisp and Smalltalk, as it is not easy to introduce explicit parallelism in their computational models. There exist however examples of work introducing explicit parallelism into functional languages, e.g. [HALSTEAD85], and object-oriented languages, e.g. [AMERICA86]

1.3.3 Multi User support

As mentioned in the previous section multiple users are easily supported in a language with explicit parallelism. But it is not enough to allow multiple users to exist, there should also be some support so that they can cooperate without making a mess for each other. This is achieved by various methods on the different levels:

- Programming languages provide synchronization primitives, and these must be applied correctly by the program to achieve the desired degree of mutual exclusion.

- Data bases provide transaction facilities, ensuring the atomicity of transactions. This is achieved either by implicit locks on objects, or by some form of optimistic concurrency control.

- Operating systems provide the possibility to explicitly lock objects.

In a software development environment the transactions are usually very long. This makes the use of implicit locks prohibitively restrictive with regards to parallel work. Optimistic concurrency control is also out of the question as the transactions include interaction with the user. Rather than any strict locking, there is actually more a need for a notification facility, and a possibility to cooperate on large objects. One simple and elegant solution is provided by Emacs, which notifies users visiting the same file.

1.3.4 Programming styles and paradigms

The language should be flexible enough to permit and support different programming paradigms. So, even if Nel does not directly provide all policies of functional, logical or object oriented programming, the basic mechanisms are present, i.e.:

- Functional programming requires first class functions, and the possibility to pass all kind of objects as parameters and results.
• Logical programming requires unification, and a possibility to save the rest of the computation for later backtracking.

• Object-oriented programming requires a flexible handling of environments (records).

All these primitives are present in Nel. Furthermore all objects should be treated equally, i.e. be first class.

1.3.5 User interaction

A user needs to interact with the system. In Lisp and Smalltalk environments this is achieved by an interpreter which interprets commands written in Lisp or Smalltalk. Conventional operating systems like Unix have a procedural interface, on top of which command interpreters are built (shells). The integrated approach taken in Lisp and Smalltalk is superior.

Another important component of every programming environment is the editor in which programs are written. Some editors are becoming quite powerful, relieving the user from ever leaving the editor. The Emacs editor is a prime example of these new generation of editors. Another interesting aspect of the Emacs editor is that it is equipped with a complete programming language for the user to customize the editor. This makes the editor very flexible, but forces the user to learn yet another language. However, making all interaction with the system through an editor relieves the user from learning different “editing commands”. So the goal is to make the editor language the same as the programming language.

1.3.6 Persistence

Persistence is the ability for objects to persist independently of the function / process which created them. Orthogonality with regards to persistence implies that all kinds of objects should be able to persist, so there should be no distinction between objects internal or external to a function.

This requirement is to a certain extent achieved in Lisp and Smalltalk if we disregard the problem of moving objects between different “images”. With the incorporation of parallelism in the language, multiple users can exist in the same “image”, and thus share the objects.

1.3.7 Versioning

The system should be able to record several versions of the same object, and help the user manipulate these versions. In the VMS\textsuperscript{4} operating system [KENAH84] a revision chain versioning is incorporated into the file system. Versioning systems

\textsuperscript{4}VMS is a trademark of Digital Equipment Corporation.
are crafted on top of Unix by RCS/SCCS. NSE [COURINGTON89] represents an
interesting approach to a language independent environment containing support
for SM. One central idea in NSE is the concept of workspaces which are committed
and merged under user control.

1.3.8 Naming

When the number of objects grows, there is a need to organize these objects.
Hierarchical structures like file systems or nested records are the usual means.
These structures should be able to contain all kinds of objects, which is the case
for Lisp, but not Smalltalk where the classes are stored in a flat namespace (the
Smalltalk pool). PCTE/OMS [PCTE86] is an example of an extension to basically
hierarchical and poorly typed file systems like Unix. These systems extend the
PITL structures towards those found in PITS.

The ability to bind names to objects should also be flexible. Some example of
bindings:

- Totally dynamic, which requires a search for the name. Examples are the
  search for methods in Smalltalk and files in Unix.

- “Semi” dynamic (indirect), where the search for a name is replaced by one
  or more levels of indirection. Examples are array access, and accessing at-
  tributes of objects in Simula.

- Static, which is represented by e.g. FORTRAN’s scalar variables.

1.3.9 Typing

The system should support strong typing. But as any object may persist, the
possibilities of static type checking are limited.

Type information is also useful to restrict the values which can be “stored” in
an object. The user should be able to specify the type of an object as precisely
as wanted. Thus the type will serve as a kind of restriction on the values of the
object. In dynamically typed language like Smalltalk and Lisp it is not possible
to restrict an object to contain only a limited number of values. Statically typed
object oriented languages like Simula allow a pointer declared to point at a class,
$X$, also to point at an object of any subclass of $X$. Unix has a “one type suits them
all” approach to typing, where the byte file is the universal type. This is in effect
an untyped environment, where each program is free to interpret the contents of
the file as it wants. PCTE/OMS and Galileo [ALBANO85] are examples of PITL
systems where a stronger concept of type has been introduced.

There should be some means to convert from most values or types to a “typeless”
text representation and vice versa. This is important for external devices, and
could be achieved by a textual input / output for all values. This is similar to the
external representation of Lisp function / data.
Conventional programming languages like Pascal can only do input / output on a limited set of types. This is also tied up with the static type checking, where a more flexible input would require parsing and type checking in the input routine.

1.3.10 Information hiding

Information hiding and access control are important in multi user environments. The process creating an object should be able to restrict the use of this object from other processes. Information hiding constructs have also been introduced in programming languages to make the implementation independent of the use of a module. The disadvantage of these constructs is that the information hiding is solely enforced by the compiler.

1.4 Some similar systems and related work

The scope of this work implies that it relates to a wide spectrum of work concerning programming paradigms and environments. In this short section it can only be tried to relate it to a few other approaches.

1.4.1 Unifying approaches

The need to unify the different languages and find some common denominator soon became apparent. The first attempt to make an uniform intermediate language to which all languages could be translated, and for which one could generate machine code for all different machines, was UNCOL (Universal Computer Oriented Language) [STRONG58].

Landin published two papers [LANDIN65] and [LANDIN66] describing a significant part of Algol 60 in an augmented lambda calculus, and suggesting that most languages were only syntactic sugar over some basic mechanisms. The problem with the lambda calculus is obviously its lack of built in abstractions, and much of the research on programming languages can be seen as the search for the right abstractions (syntactic sugar :-). The context free syntax [CHOMSKY56] was widely established as the standard means to describe the syntax of a programming language [NAUR63], but until the late sixties no real breakthrough in describing the semantics had come. Then in the late sixties three approaches emerged;

- Strachey and Scott [STRACHY66] [SCOTT70] developed the denotational approach mapping the syntactic structures into functions on domains representing e.g. store and environment.

- Floyd and Hoare [FLOYD67] [HOARE69] developed the axiomatic approach describing the meaning of statements with pre and post conditions, i.e. if the
state satisfies the pre-condition before the statement, then it will satisfy the
post condition after the statement is executed.

- Knuth [KNUTH68] developed the attribute grammars, which augments the
syntax tree with attributes, which are propagated and evaluated by local
rules at each node in the syntax tree yielding the meaning of the program.

The expressive power of these three formalisms are equivalent, but they have had
different use. Denotational semantics have been the primary vehicle for language
descriptions, and given a deeper insight into the fundamental concepts of program-
ming language design. Axiomatic semantics have been used mostly in program
verification, and attribute grammars have had their main use in specifications of
compilers and syntax editors.

1.4.2 Ada

The Stoneman report [STONEMAN80] gives the requirements for the Ada Pro-
gramming Support Environment (APSE), which shall include three principal fea-
tures; a data base, a user interface and a tool set. APSE is divided into two lower
levels, Kernel APSE (KAPSE), and Minimal Tool Set (MAPSE). Thus Ada/APSE
is an example of a language specific program support environment.

KAPSE shall provide a database and run-time system to execute Ada programs.
MAPSE provides the compiler and some support tools, e.g. editor, debugger,
linker/loader, JCL interpreter and configuration management. The compiler also
includes static and dynamic analysis tools, e.g. def-set-use and flow analysis.

The full APSE should provide life cycle and project support, and can be tailored
to specific development methods, whereas MAPSE is method independent. APSE
should include a documentation system to support the production and control of
other parts of the software product than the program code.

There has also been work on an common intermediate representation for Ada
programs, i.e. DIANA [GOOS83], which shall facilitate the interconnection of tools.

The requirements outlined in the Stoneman report indicate a loosely coupled en-
vironment for one specific language. By loosely coupled is meant that the tools
are rather stand-alone, and there are possibilities to include new tools, i.e. the
environment is open-ended.

1.4.3 Cedar

The Cedar language is a strongly typed, compilation oriented language, for which
an interactive experimental programming environment has been built [TEITEL-
MAN84]. This is contrary to Lisp and Smalltalk environments which support
interpreted dynamically typed languages. Even if Cedar is a compilation oriented
language, an interpreter is included to facilitate debugging and let Cedar work as
an interactive language.
Cedar integrates aspects of the PITS and SM by using the System Modelling
Language [LAMPSON83] to model the following aspects of the system:

- The versions of various modules that make up the system. The versioning
  model is quite simple, as it is based on time stamps, i.e. only modelling a
  revision chain.
- The interconnection between modules.
- Information on how to compile and load the system. This is based on tran-
  sitively recompiling every object module outdated by a source module. The
  distinction between interface and body restricts the recompilation to the
  body if the interface has not changed.

Note that except for the version selection these are superfluous tasks in an in-
terpreted environment like Lisp or Smalltalk as there are no out of date object
modules.

1.4.4 Related work

The idea of integrating PITS, PITL and SM is not new.

In [JONES77] they argue to narrow the gap between language systems and oper-
ating systems because of the introduction of concurrency into programming lan-
guages. This will also ease the verification of programs as they will now only
involve one system, and not both the programming language and the operating
system.

In [HEERING85] design principles for a monolingual programming environment is
described. They want to unify command, programming and debugging languages
into one, and their main requirements for such a language are:

- Programs and procedures are the same.
- Types of permanent and local objects are the same. New types may be added
  by the user.
- Procedures and type definitions are objects.
- A program can be switched between batch and stepwise execution.
- “Elastic” type checking, i.e. the type checking is done as soon as possible,
  and may have to wait until run time.
- Events can be associated with conditions, i.e. general triggers, similar to
  conditions on variables in debuggers.
- Side effect recovery by a try/probe construct, which is not easily integrated
  into an interactive environment.
The main difference between the ideas in this work and the Nel environment is that Nel support multiple users and incorporate ideas from software maintenance.

In [GABRIEL88] preliminary requirements for a common prototyping system are presented. It turns out that these requirements are to a large degree the same as those presented in section 1.3.
Chapter 2

The Nel language

2.1 Introduction

This chapter contains an informal introduction and discussion of the Nel language. Nel is an imperative expression language, and includes objects which are second class in most other languages as first class objects, e.g. environments and continuations. Nel has replaced the traditional notion of type and value with the notion of maximal and actual value\(^1\) of a denotable value (location), where all the values are collected in a lattice. Furthermore Nel has multiple values, unification, access control and concurrency. Languages with similar constructs are proposed in the literature, e.g.:

- Environments are records, classes and modules. Inheritance is a special form of concatenation of environments.

- Continuations are first class objects in Scheme, and appear in different disguises as labels, coroutines and exceptions in other languages.

- Multiple values give the ability to pass around multiple values instead of only one, without making a new value of it. A simple form is to pass multiple parameters to a function. In Lisp and Beta [KRISTENSEN83] multiple values can also be returned.

- Unification, i.e. finding the greatest lower bound is present in logical languages like Prolog, and in the restricted form of pattern matching in functional languages.

- Concurrency is explicitly present in languages like Algol 68 and Ada, and can also be implicitly introduced like in Concurrent Prolog [SHAPIRO87].

- A type can be seen as a restriction on the values of a location, and are present in one form or another in most languages.

\(^1\)Because Nel lacks the distinction between the domain of types and values, these two words are used as synonyms.
Even with all these features Nel is semantically not very complex. This is mainly due to the orthogonal use of all concepts, and the emphasis on providing mechanisms and not policies.

Most other languages have made a trade-off between efficiency and expressibility, and are mainly biased towards PITS (programming-in-the-small).

Since one of the main points in this work to discuss how we can support aspects of PITL and SM (software maintenance) through the use of orthogonal constructs in a PITS language, maximal expressibility is required for these constructs. The expressiveness of the constructs in Nel is elaborated through the examples in chapter 3.

Nel and its integrated programming support environment is thus intended to cover the tasks normally trusted to operating systems, debuggers, system modelling and version control systems for other languages. The Nel programming support environment is described in chapter 4.

A definition of Nel in a denotational semantics style is given in chapter 5.

The rest of this chapter

The rest of this chapter is organized as an informal introduction to Nel. Section 2.2 describes the primitive values and their operations. In section 2.3 the value constructors are described. Assignment and access control is explained in section 2.4. Control flow, both sequential and parallel, is treated in section 2.5. Lastly, section 2.6 explains the input/output facilities in Nel.

Typographical conventions

In this chapter the following typographical convention will be used: typed text represents syntactic constructs, while italics represents semantic constructs.

2.2 Primitive values

All constructs in Nel are expressions and return a denotable value, $DV_{al}$, which can be compared with a location and can be treated as a first class object. The values can be divided into primitive values and constructed values. Primitive values are values built into Nel, whereas value constructors are used to make constructed values. The primitive values are treated in this section, whereas the value constructors are discussed in section 2.3.

A general expression is written $E$, and comments are written inside /* */. Comments can be nested.
2.2.1 Simple values

Simple values are values which are represented as singleton (or empty) sets in the set model. In Nel numbers (3, 23, 18, ...), booleans (TRUE and FALSE), and characters ('A', 'f', '3', ...) are simple values. The non printable characters are written as e.g. \101, where the number is the character code. Furthermore VOID (the empty set) and NIL (the empty pointer) are regarded as simple values.

The following operations are defined on the simple values:

<table>
<thead>
<tr>
<th>Values</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>+</td>
<td>binary, returns a number</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>binary, returns a number</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>binary, returns a number</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>binary, returns a number</td>
</tr>
<tr>
<td></td>
<td>MOD</td>
<td>binary, returns a number</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>binary, returns a boolean</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>binary, returns a boolean</td>
</tr>
<tr>
<td>Booleans</td>
<td>AND</td>
<td>binary, returns a boolean</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>binary, returns a boolean</td>
</tr>
<tr>
<td></td>
<td>NOT</td>
<td>unary, returns a boolean</td>
</tr>
<tr>
<td>All</td>
<td>=</td>
<td>value equality, returns a boolean</td>
</tr>
<tr>
<td></td>
<td>&lt;&gt;</td>
<td>value inequality, returns a boolean</td>
</tr>
<tr>
<td></td>
<td>==</td>
<td>location equality, returns a boolean</td>
</tr>
</tbody>
</table>

Note that there are only one kind of numbers in Nel, these are rather arbitrarily chosen to be 64 bits around 0. Exceptions are raised for overflow and divide by zero.

2.2.2 Set values

Set values are values represented as a set with more than one element in the set model. In Nel there are the set of all booleans, BOOL (TRUE and FALSE), the set of all numbers, INT, and the set of all characters, CHAR. Furthermore Nel has the set of all values, DIOV (VOID backwards :-).

The value lattice

All values are collected in a value lattice. The partial ordering (\sqsubseteq) among the values can either be regarded in a set model, or an information contents model, where the values in the upper part of the lattice represent bigger sets, or are less specific than those in the lower part. The position of each value in the value lattice will be given when the value is introduced. The value lattice is used to determine when an assignment is legal (the \sqsubseteq-relation must hold between the source and destination), and unification which is the greatest lower bound (glb) of the two values.

The value lattice for the values introduced so far is given in figure 2.1.
The partial order on the values is written \( a \sqsubseteq b \), and means that \( a \) is more specific than \( b \), i.e. we know more about \( a \) than \( b \). The most undefined value is DIOV of which we have no information, and the most defined value is VOID of which we have complete information.

The (non-recursive) values can also be regarded as sets. We then regard the \( \sqsubseteq \)-relation as the usual subset relation. DIOV is the set of all values, and VOID is the empty set. For recursive values there is no such simple interpretation, and they have to be defined as elements in recursive domains [SCOTT76].

The LEQ operator is defined to check if two values are in the \( \sqsubseteq \)-relation. It is infix binary and returns a boolean.

The sets denoting some of the values introduced so far are:

<table>
<thead>
<tr>
<th>Syntactic name</th>
<th>Set interpretation</th>
<th>Semantic name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOID</td>
<td>{} or (\emptyset)</td>
<td>Void</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>1</td>
</tr>
<tr>
<td>TRUE</td>
<td>{true}</td>
<td>True</td>
</tr>
<tr>
<td>BOOL</td>
<td>{true, False} = {true} \cup {false}</td>
<td>Bool</td>
</tr>
<tr>
<td>INT</td>
<td>{minint, ..., maxint}</td>
<td>Int</td>
</tr>
<tr>
<td>'c'</td>
<td>{'c'}</td>
<td>c</td>
</tr>
<tr>
<td>CHAR</td>
<td>all ascii characters</td>
<td>Char</td>
</tr>
<tr>
<td>DIOV</td>
<td>INT \cup CHAR \cup BOOL</td>
<td>Diov</td>
</tr>
</tbody>
</table>

There are two prefix unary operations to check if a value is a singleton set or if the value contains a VOID component. These are GROUNDED and VOIDED, both returning a boolean. VOIDED is deep, i.e. it follows pointers and environments. Grounding is
shallow, e.g. an environment value can contain set valued components.

2.2.3 Declarations, denotable values and store

When the syntactic expression for a primitive value (e.g. INT) is evaluated, a location, (e.g. 2419) in the store is allocated. This location is given both initial (actual) and maximal value (int). The location (2419) is called the denotable value which is what is returned from the evaluation of the syntactic expression.

![Diagram of DVal and Store](image)

**Figure 2.2: The result of evaluating INT**

To give a name to the denotable value the E I construct is used. The name I is then bound to the denotable value returned from evaluating E, i.e. making an alias for it. The name will be visible throughout the current scope. A name can consist of any combination of letters, digits, ' (backquote) and _ (underscore), started by a letter. Keywords which are either written with only capital or only small letters are not allowed as names.

To make declarations local to a given expression the BEGIN E END construct is used.

![Diagram of Environment and Store](image)

**Figure 2.3: The result of evaluating INT a**

If a previously bound name is redeclared, the new declaration will hide the old in the current scope. UNDEF I is used to hide a name without redeclaration in the current environment.

The binding from names (identifiers) to denotable values is called the *environment*, and the binding from denotable value (actually the location) to the (actual) value is called *store*.

A simple example of some declarations:

```plaintext
INT a;
BEGIN
```
DIOV b; /* a and b are visible */
UNDEF a;  /* only b is visible */
b c     /* c is an alias of b, b and c are visible */
END     /* a is visible */

A unary operator ISDEF I is applied to check if an identifier is defined in the current environment.

## 2.3 Value constructors

Composite values can be made from primitive values by value constructors. There are constructors for making pointers, multiple values (array-like values), environments (records), unions and intersections (unification). In section 2.5 program text values like functions and continuations are treated.

### 2.3.1 Pointers

Pointers make the denotable values first class objects, i.e. include the denotable values in the storable values. To build a pointer\(^2\) to a value, the unary prefix operator REF is used. Prefix unary DEREF is used to follow a pointer, and NIL is the empty pointer.

The following example shows some pointer values, and their place in the value lattice:

### 2.3.2 Multiple values

Multiple values are created by the ,-operator. Multiple values can be regarded as a list (or array) of denotable values. Nested multiple values are constructed explicitly by building references.

Two examples:

\[
\text{INT,INT,BOOL } x; \\
\text{(INT, REF } x) \text{ y}
\]

Note here that \(x\) maps to a list of denotable values, i.e. a multiple value. \(y\) is a “nested” multiple value.

To get a specific value out of a multiple value, the \(E[E]\) construct is used. The last \(E\) should evaluate to a positive number, and the corresponding value in the multiple value is returned. If no such value exists, an error occurs. The index of multiple values starts at 1. To get the number of values in a multiple value the LEN operator is used. Example:

\(^2\)or reference, which will be used as synonyms
Figure 2.4: The value lattice for pointer values

```
INT, INT, FALSE, BOOL x;
\[x[\text{LEN } x]\]  /* BOOL */
```

We can bind identifiers to different parts of the multiple value by the E I: I: . . .: I syntax, where the last I will bind to the rest of the multiple value. It is an error if there are more identifiers than there are values in the multiple value. An example:

```
INT, INT, INT hd:tail;
```

Note that the multiple identifiers can be viewed as a destructor for the , constructor.

By enclosing a sequence of characters in double quotes Nel provides a shorthand for making multiple character values. E.g. "abc" is the same as 'a', 'b', 'c'.

Note that concatenating values into a multiple value does not make any new values. The argument values are shared with the components of the multiple value, i.e:

```
INT y; INT x;
y.x \text{ c};  /* c[1] and c[2] are aliases for y and x */
```

Operations like +, =, AND, := and ~ (unification) work on multiple values of the same length.

The reason for not making multiple values directly storable values is that it creates problems in connection with assignment to a denotable value with maximal value DIOV. Should the whole multiple value be assigned, or just some part? This problem
is avoided by having the multiple values as multiple denotable values. E.g. \((\text{DIOV } x, \text{ BOOL } y) := \text{ TRUE, FALSE}\) assigns \text{TRUE} to \(x\) and \text{FALSE} to \(y\). \text{DIOV} := \text{ TRUE, FALSE}\) gives an error, whereas \(\text{DIOV} := \text{ REF (TRUE, FALSE)}\) is perfectly legal.

A reference \(\text{xref}\) to multiple value \(xmv\) is a subtype of reference \(\text{yref}\) to a multiple value \(ymv\) if the length of \(xmv\) is greater or equal that of \(ymv\), and the maximal value of each component of \(xmv\) is a subtype of the corresponding maximal value in \(ymv\). Some examples are shown in figure 2.5.

---

**Figure 2.5: The value lattice for pointers to multiple values**

---

### 2.3.3 Environments

An environment value is a collection of declarations, and corresponds to a record. The declarations are local to the environment, and must explicitly be made visible. To make an environment the declarations are enclosed in \(\{ \} \), i.e. \(\{ E \}\). Opening of an environment is done by the binary infix operator \(\cdot\), i.e. \(E \cdot E\).

The ordering on environments is based on the information contents. The empty environments is the most general, i.e. least information. An environment, \(zenv\), is not less specific than another environment, \(yenv\), if \(zenv\) contains all the names of \(yenv\), and these names are bound to multiple denotable values of not shorter length in \(zenv\) than in \(yenv\), where each denotable value has a not less specific maximal value in \(zenv\) than in \(yenv\). The order of declaration is thus without importance.

A simple example:

```plaintext
{ INT a; INT b } c;       /* construction */
c.a                      /* selection */
```

Figure 2.6 shows the value lattice for some environments.

There is always a default current environment where names are looked up. \text{THISENV}\ makes a denotable environment value of the current environment. Furthermore
there is a binary infix operator, & , to concatenate two environments. The declarations in the last one will hide those in the first.

Examples:

{INT a} b;
{INT a} c;
b&c d;        /* d.a alias c.a and b.a is not visible from d */
THISENV.b    /* same as b */

Note that when a new environment is made with the & operator, its identifiers will be mapped to the denotable values of the old environments.

Undefined names will not continue to hide names when environments are concatenated, e.g.

{INT x}&{UNDEF x}

makes the first x visible.

Records opened by with and . in Pascal can be modelled by environments;

rec.a          /* rec.a */
(TTHISENV&rec).( /* WITH rec DO */
    E)          /* Stm; */

Note the similarity between pointers and environments with one component, i.e.

REF INT pointer; {INT cont} pointer; /* building */
DEREF pointer:= 7; pointer.cont := 7; /* selection */
pointer := REF INT; pointer := {INT cont}; /* changing */

21
2.3.4 Union

The binary \texttt{UNION} operator makes a value which is either of the values of the two operands. In the set model it is interpreted as the union of the two argument sets. The constructed values have their natural place in the value lattice. Some examples of union values and their place in the type lattice is given in figure 2.7 below:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{union_lattice.png}
\caption{The value lattice for some union values}
\end{figure}

\texttt{UNION} is deliberately chosen not to be the disjoint union, as in many other programming languages. The reason is that at the moment of construction we do not have any more information about the actual value of the union. \texttt{DIOV} is the union of all values in \texttt{Nel}.

The \texttt{UNION} operator works on both the maximal and actual values of its argument, making a new denotable value which has as maximal value the union of the maximal values of its arguments, and as actual value the union of the actual values.

\texttt{UNION} is shallow, i.e. the union of pointers and environments will share structures with the arguments. An example with environments:

\begin{verbatim}
{INT a} x1; {INT a} y1;
(x1 UNION y1) u1;
x1.a := 3; y1.a := 3;

INT x2; INT y2;
(x2 UNION y2) u2;
x2 := 3; y2 := 3;
\end{verbatim}

\texttt{u2} is unaffected by the assignment, whereas \texttt{u1}'s components are. This is however of minor importance, as there is no way to use one component of a union value.

The \texttt{GROUND}ed operation introduced in section 2.2.2 is useful for checking if a union value is one of its components.

2.3.5 Unification

\texttt{Nel} also provides unification, which is a binary operator \texttt{\sim}, returning a maximal and an actual value which are the greatest lower bound of the actual and maximal values of the operands respectively. Unification is deep.

Examples:

\begin{verbatim}

\end{verbatim}
INT ~ 3 /* 3 */
TRUE ~ 4 /* VOID */
DIOV ~ INT /* INT */
3 UNION 4 ~ 4 UNION 5 /* 4 */
{ INT a } ~ INT /* VOID */
{ INT a } ~ { INT b } /* {INT a; INT b} */
{INT a; INT b} ~ {DIOV a} /* {INT a; INT b} */
{3 a} ~ {4 a} /* VOID */

Environments will unify to an environment as long as their names are unifiable, only when the names are not unifiable will VOID be the result. As the UNION operator can be interpreted as the union operation on the set model, the unification operation can be interpreted as the intersection.

The operators GROUNDED and VOIDED introduced in section 2.2.2 are useful for checking the result of unifications.

### 2.4 Assignment and access control

#### 2.4.1 Assignment

Nel is a strictly typed language and assignment is only allowed when the maximal value of the destination is less specific than the actual value of the source. There are two assignment operators := and =:, so that assignment can work both ways. Assignments to pointers and environments are shallow. The denotable value returned from both assignments is the newly assigned one.

Note that only the actual value is changed by assignment, the maximal value remains constant.

Some simple examples:

\[
\begin{align*}
\text{INT } a & := 3 =: \text{INT } b; \\
\text{INT } a & := \text{INT } b := 3; /* the same */
\end{align*}
\]

\[
\begin{align*}
a & := 4; /* OK */
\end{align*}
\]

\[
\begin{align*}
a & := \text{TRUE}; /* not allowed */
\end{align*}
\]

Because assignment is shallow for pointers and environments the maximal values of the components of the source are checked against the destination. Some examples:

\[
\begin{align*}
\{\text{INT } a\} & := \{\text{DIOV } b:3\}; /* type error */
\{\text{INT } b\} & := \{4 \text{ b}; \text{DIOV } c\}; /* OK */
\text{REF INT} & := \text{REF } (\text{DIOV } := 4); /* not allowed */
\end{align*}
\]

The LEQ operator introduced in section 2.2.2 checks for assignability.
2.4.2 Copies

The unary operators COPY, VAL and TYPE makes a new value based on the operand as follows:

<table>
<thead>
<tr>
<th>Operand</th>
<th>maximal value</th>
<th>actual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td>( v_{\text{max}} )</td>
<td>( v_{\text{act}} )</td>
</tr>
<tr>
<td>VAL</td>
<td>( v_{\text{max}} )</td>
<td>( v_{\text{act}} )</td>
</tr>
<tr>
<td>TYPE</td>
<td>( v_{\text{act}} )</td>
<td>( v_{\text{max}} )</td>
</tr>
</tbody>
</table>

The copies are deep, i.e. following and making new copies of the contents of pointers and environments.

When there are aliases involved the result of the copy operations will have the same aliases as the arguments, e.g.:

\[
\text{COPY } \{ \text{INT a; a b := 3} \} \\
\text{VAL } \{ \text{INT a; a b := 3} \} \quad /* \{ \text{INT a; a b} \} */ \\
\text{TYPE } \{ \text{INT a; a b := 3} \} \quad /* \{ \text{INT a; a b} \} */
\]

VAL is the source of recursive maximal values (types) in Nel. Recursive actual values are easily made with environments and pointers, e.g.

\[
\{ \text{DIOV x} \} \ a; \\
a.\ x := a;
\]

\[
\text{REF DIOV b;} \\
\text{DEREF b := b;}
\]

but they will not have recursive maximal values. If we use the VAL operator on either \( a \) or \( b \) we get a recursive maximal value.

2.4.3 Access control

Initially a denotable value in Nel can be both changed (written to) and bound to another name. To protect the denotable values two constructs to create aliases are provided in Nel:

- The prefix unary CONSTANT operator, which makes an alias through which the denotable value cannot be changed.

- The unary LIMIT operator, which makes an alias through which the structure of the denotable value cannot be revealed, i.e. copied.

Examples:
INT a;
CONST a b;
LIMIT a c;
  a := 3; /* OK */
  b := 5; /* wrong, cannot be changed via b */
  c := 5; /* OK */
REF c; /* OK */
  c d; /* OK */
  c := c+5; /* wrong */
  a := c; /* wrong */

We can use the LIMIT operator to achieve a kind of data confinement, i.e. we pass
a limited denotable value to a procedure, the procedure can use it and pass it
further, but when it returns we invalidate all these shallow copies by assigning a
VOID to the parameter. E.g.:

LIMIT (DIOV nonlimit := original) parameter;
nottrustedfun(parameter);
original := nonlimit;
nonlimit := VOID;

A shallow copy is made by the assignment to DIOV to avoid changing the original
when the access to the value is eventually withdrawn by resetting the copy.

The only allowed operations on a limited denotable value are assignment to, build-
ing a reference, making an alias, passing as a parameter, and use as a function, a
continuation or an environment ³

Both CONST and LIMIT make shallow copies. I.e. an alias for the argument is
provided. This alias is on the denotable value level. When both the original
denotable value and a CONST alias appear in unification, the result will also have
this structure. E.g:

{DIOV a; CONST a b} ~ {DIOV a; INT b} /* {INT a; CONST a b} */

Both CONST and LIMIT are shallow, i.e. a component of a limited environment can
be copied. This is different from the access mechanisms in e.g. Unix, where read
only on directories is deep. The semantics of the shallow operations are simpler,
and to emulate the deep semantics is not too complicated.

There are two unary operators ISCONST and ISLIMIT to check if a denotable value
is either constant or limited.

³There is a breach in the confinement provided by LIMIT for environments, i.e. env :=
limenv.(THISENV) which amounts to copying the limited environment. This could be avoided by
propagating the information on what kind of environment one is using, and disallowing THISENV
when using a limited environment. With this information we could also make LIMIT and CONST
deep for environments by propagating this information into the denotable values which are accessed
through the environment.
2.5 Control flow

In this section control flow aspects of Nel are discussed. This includes sequential control flow in section 2.5.1, function values and their use in section 2.5.2, continuations in section 2.5.3, and in section 2.5.4 concurrency.

2.5.1 Sequential

Sequential control flow is provided by the usual means: E;E and E >> E for sequencing. E >> E means that the result from the first E is passed to the second, whereas in E;E the intermediate result is voided, i.e. E >> VOID >> E.

There is also IF E THEN E ELSE E FI for choice, and WHILE E DO E OD for iteration. Note that the IF-construct is an open scope, whereas WHILE is closed, both between each iteration, and after the loop. The ELSE E part of the IF construct is optional, in case it is left out ELSE VOID is assumed.

The value returned from a WHILE-loop is the value computed by the last iteration of the body. The value into the body is the value before the loop for the first iteration, and the value of the preceding iteration for the rest.

The STOP construct ends the execution of this process.

There is a construct -> to get the current denotable value. Examples of the usefulness of -> and >> are the following constructions:

\[
\begin{align*}
\text{INT } i & := a >> \quad \text{INT } i := 0; \\
\text{WHILE } (i < b) \text{ DO } & \quad \text{INT } >> \\
\quad \rightarrow \text{ UNION } (i := i + 1) & \quad \text{WHILE } (i := i + 1) <= b \text{ DO } \\
\quad \text{OD } c & \quad \rightarrow , \text{ INT } \\
\text{OD } d & \\
\end{align*}
\]

Here c gets bound to a union of integers, i.e. the subset a..b in Pascal, and d is bound to a multiple value, where each value is INT, i.e. an array.

The priority of operators are given in A. (E) is used to override these priorities.

2.5.2 Functions

A function is a piece of code which is stored as a value. To make a function value the FUN E NUF construct is used. A function is called by the E(E) construct. Functions in Nel are statically scoped, i.e. the environment on the place of declaration is used to bind global variables in the function.

Examples:

\[
\begin{align*}
\text{FUN } \rightarrow & \quad a : b; \\
\text{FUN } \rightarrow & \quad =: (\text{DIV} \ a, \text{INT} \ b);
\end{align*}
\]
INT i := a >>
WHILE (i < b) DO
  UNION (i:=i+1)
OD

NUF subset;
subset(23,34) c;

a(VOID) >>
WHILE (i:=i+1) <= b DO
  a(VOID)
OD

NUF array;
array(FUN INT NUF,34) d;

Note that there is no explicit receiving of parameters, the current (denotable) value has to be picked up by the -> construct. The subset function emulates call by reference, and the array function call by value.

The name of the function is not available inside the body (FUN E NUF), so to make a recursive function we have to assign it to a named DIOV variable, e.g.:

DIOV fac :=
  FUN -> n;
  IF n = 0 THEN
    1
  ELSE
    n*fac(n-1)
  FI

2.5.3 Continuations

When calling a function the current continuation (rest of the program) is passed along as an implicit parameter which is implicitly used at the end of the body. The E//(E) (call-with-current-continuation) construct is used to pass the continuation as an explicit parameter to a function. The continuation is activated with the, E//(E) construct. The result passed to the continuation is given as parameters to an ordinary procedure. The procedure called with the current continuation can also receive more parameters with the syntax E//( E).

This is an improvement from Scheme where it is difficult to pass extra parameters along with the continuation back to a continuation. E.g. in Simula both the environment and the continuation are implicitly returned from a detaching object. The use of the same syntax for call of functions (which are returning) and continuations (which are not) is also misleading in Scheme.

Continuations can be used to model most flow of control features, two relatively simple examples are labels and coroutines:

FUN -> NUF label; /* just return the input value */
label//(E) a;    /* make a label, i.e. the continuation */
a//(a)          /* goto a */
FUN -> det;
    /* do something */
  det(/* */) det;
    /* detach */
  /* do something more */
NUF cor;
    /* a coroutine */
cor(/* */) res;
    /* call cor, bind rest to res */
res(/* */) res
    /* resume cor */

Function and continuations are values. They are all distinct, and are more specific
than DIOV and less specific than VOID.

The static environment of functions are obtainable by STATENV E, which makes a
denotable value containing a copy of the static environment. The static environ-
ment cannot be obtained from LIMIT denotable values.

2.5.4 Concurrency

Nel has a simple form of concurrency, i.e. splitting a process by FORK and synchro-
nization with semaphores. The reason is to introduce multiprogramming, not to
model concurrent language constructs.

Each process in Nel has a unique process identifier which has the constant alias
MYPID. All process identifiers are distinct denotable environment values. To make
two concurrent processes the FORK construct is used. FORK returns the process
identification of the twin process, i.e. the other process. The two processes share
the variables declared before the fork, but variables declared after become private.

An example:

    IF MYPID = FORK THEN
      /* FORK = MYPID will not work */
    /* clone */
    ELSE
      /* original */
    FI

This works properly because MYPID is used to find the process identifier before the
fork (the original), and will thus be used in both the original and the clone. If we
interchange the evaluation both processes will execute the else branch.

The process identification is a constant environment consisting of:

- dval, a constant pointer to the last denotable values of the process.
- env, a constant denotable value containing the last environment of the pro-
  cess.
• cont, a constant denotable value containing the continuation of the process. Note that cont is statically scoped. E.g. executing the continuation of the halted process in the current process:

    pid.cont//<(DEREF pid.dval);

• pidnr, the process number of this process. This is constant, e.g. CONST 34 pidnr.

• std, a constant environment containing pointers the standard input (in), standard output (out) and standard (err) device variables for this process. Input and output are explained in section 2.6.

• status, a pointer to a constant string containing the status of the process.

• pid, a constant function used to make the process identifier unique, and not interchangable, i.e. a process cannot construct an environment, and pass it as as a process identifier.

The first three fields are VOID when the process is active (running or ready to run), and will be set when the process is halted, makes an error, or finishes naturally. They will also be set when the process is waiting for a semaphore or the termination of another process. Changing these fields has no effect on the original state of the process, but the behaviour can be modified indirectly by changing the store through dval and env. When a process executes a limited function or continuation the first three fields are void even if the process is waiting on a process or semaphore, and the process can not be halted.

The construct HALT(E) halts the processes denoted by E. It will return FALSE if the process was already halted, and TRUE if it is halted now. HALT returns VOID if the process was executing a limited function. The process executing a limited function can be halted by LIMITHALT(E1,E2). E1 evaluates to a non limit access to the limited function which is executed, and E2 is the process identifier.

A halted process can also be restarted by RESTART(E). It will return TRUE or FALSE depending on whether the process could be restarted, i.e. only halted processes can be restarted. The restarted process uses the values in dval, env and cont.

There is also a construct SUSPEND with four parameters, cont, dval, env and status, which stops this process and updates the process identifier with the parameters. The following example illustrates the connection between HALT, STOP and SUSPEND

```
FUN
E1 >>
HALT(MYPID) >> SUSPEND(/-(REF (->)). THISENV.
                    REF CONST ("halted");
E2           TRUE >> E2
NUF           NUF
```
A construct \texttt{WAIT(E)} is provided to let the current process wait on the termination of the processes denoted by E. \texttt{WAIT} returns instantaneously if the processes have already terminated. The return value of \texttt{WAIT} is \texttt{TRUE} if the waited for process finished, i.e. \texttt{STOP}, \texttt{FALSE} if it was halted, and \texttt{VOID} if the current process was explicitly halted and resumed. \texttt{WAITING(E)} gives a multiple denotable value of the process number of all processes waiting for the termination of the process denoted by E.

Processes are collected in a hierarchy, and \texttt{PS(E)} will give a multiple denotable value of the process identifiers of all processes cloned from E including itself.

To synchronize processes semaphores, \texttt{SEM} with \texttt{P(E)} and \texttt{V(E)} are provided. Semaphores appear like process identifiers as constant environments consisting of:

- \texttt{semnr}, a constant integer denoting the unique number of this semaphore.
- \texttt{sema}, a constant function used to make this semaphore unique.

There is also an operation \texttt{SEMAWAIT(E)} to get a multiple denotable value of all processes waiting for this semaphore. It returns \texttt{VOID} if there are none waiting. \texttt{P} returns \texttt{TRUE} if the semaphore is passed normally, and \texttt{FALSE} if the process is halted and later resumed. \texttt{V} returns the semaphore.

Halting a process waiting for a semaphore (\texttt{P(E)}) or the termination of another process (\texttt{WAIT(E)}) removes the process from that queue. By explicitly halting and resuming a process we can let it pass a semaphore queue out of turn.

There are thus four non error states of a process:

- \texttt{waiting sem/proc} on a semaphore (\texttt{P(E)}) or another process (\texttt{WAIT(E)}).
- \texttt{active} running or ready to run.
- \texttt{halted} halted with \texttt{HALT(E)} or \texttt{SUSPEND(E)}.
- \texttt{finished} finished by \texttt{STOP}.

### 2.5.5 Exceptions

When an error occurs during execution of a Nel program, the offending operation is aborted, and the interpreter search the current environment for an identifier \texttt{TRAP}. If \texttt{TRAP} is an environment containing a name corresponding to the current exception or the name \texttt{catchall}, and that name denotes a continuation it is executed. If both the ordinary name and \texttt{catchall} are defined, the ordinary name is used.
If none are defined a standard continuation is executed. The standard exception continuation updates MYPID and halts the process. The values of the process identifier is on the same format after an exception as after HALT(E).

The built in exceptions in Nel are raised with five parameters:

- A continuation to reraise the exception. This continuation takes an environment as parameter, and a continuation is searched for in this environment. If the parameter is not an environment, the effect is the same as if a continuation is not found.

- The rest of the program, a pointer to the offending denotable values and the environment which become the values of cont, dval and env if the process is suspended.

- A pointer to a string which becomes the value of status, i.e. the name of the exception, e.g. divbyzero.

The following exceptions are defined, and will be raised by the interpreter when they occur:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Returned DVALs</th>
</tr>
</thead>
<tbody>
<tr>
<td>overflow</td>
<td>arithmetic overflow, A op B</td>
<td>A, B</td>
</tr>
<tr>
<td>divbyzero</td>
<td>division by zero, A / B</td>
<td>A, B</td>
</tr>
<tr>
<td>undefid</td>
<td>undefined identifier, name</td>
<td>REF (&quot;name&quot;),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FUN -&gt;.name NUF</td>
</tr>
<tr>
<td>nottype</td>
<td>not the expected type(^a), op A</td>
<td>A</td>
</tr>
<tr>
<td>nilpointer</td>
<td>following an empty pointer, DEREFS A</td>
<td>A</td>
</tr>
<tr>
<td>diffrange</td>
<td>multiple values of different length, A op B</td>
<td>REF (A), REF (B)</td>
</tr>
<tr>
<td>indexrange</td>
<td>indexing outside a multiple value, A[B]</td>
<td>B, REF (A)</td>
</tr>
<tr>
<td>bindrange</td>
<td>binding outside a multiple value, A I</td>
<td>REF(&quot;I&quot;), REF A</td>
</tr>
<tr>
<td>notsubtype</td>
<td>assignment is not allowed, A := B</td>
<td>A, B</td>
</tr>
<tr>
<td>constant</td>
<td>disallowed operation on CONST, op A</td>
<td>REF (&quot;op&quot;), A</td>
</tr>
<tr>
<td>limited</td>
<td>disallowed operation on LIMIT, op A</td>
<td>REF (&quot;op&quot;), A</td>
</tr>
<tr>
<td>buynewdisk</td>
<td>little real memory left(^b)</td>
<td>-&gt;</td>
</tr>
<tr>
<td>syntax</td>
<td>syntax error occurred in INPUT</td>
<td>REF &quot;text&quot;</td>
</tr>
</tbody>
</table>

\(^a\) type could be any value, e.g. INT, BOOL, FUN, giving notINT etc.

\(^b\) This condition will stop all processes which try to allocate memory, and until some memory can be garbage collected, e.g. make it unreachable by UNDEF name, or a new disk is installed, it will continue to be raised.

The process may define its own continuations to handle any of these exceptions by manipulating the TRAP environment. If the exception is not caught the process is suspended. Another process can try to fix the error and restart the process.
2.6 Input and output

Input and output in Nel is done on DIOV variables which are assigned to references to multiple character values (text), e.g. DIOV io := REF("text"). Inputting half the text from io will transfer the two first denotable values to the inputting function REF ((DEREF io)[1,2]), and make a new reference to the last two denotable values, i.e. io := REF ((DEREF io)[3,4]).

2.6.1 Parsed io

Input

The standard way of getting values into a Nel function is by INPUT(E1)(E). Here E1 evaluates to a denotable value which the text is taken from. The input text should parse to a Nel expression, and it is interpreted as the body of a Nel function with empty static environment, which is immediately called. An example:

```
DIOV stdin := REF("3 + 4", '\LF');
INPUT(stdin) + 5
/* result = 12, stdin = NIL */
```

The end of the expression is determined by <LF>. If the text does not contain an <LF> the process reads what is there and wait actively for more. Note that other processes can steal the next input if we do not protect against it.

Any syntactic error in the input will be raised as the exception syntax in the inputting process, and the text will be transferred as REF("text"). An evaluation error in the input has to be handled by the input itself. This could be achieved by passing the TRAP environment to INPUT, i.e. INPUT(stdin)(TRAP TRAP). Thus a parameter E passed to INPUT can be used in the input expression. Input and output functions can also be partially instantiated, e.g. INPUT(E) returns a function which will only read from a specific denotable value.

Output

To output something OUTPUT(E1)(E) is defined. E1 is the denotable value to receive the output. The expression E is evaluated, and the actual value of the result is unparsed, and output as text ended by a <LF>. The unparsing inserts blanks and <CR> to make the printed appearance nice. To get the type the usual TYPE operator must be used. Only one level of environments and pionters is output. Some examples of how output appears:

```
OUTPUT(stdout) out;
out(INT := 3);
/* 3 */
out('a');
/* 'a' */
```
out('a', 'b'); /* "ab" */
out(REF (INT := 4) a); /* REF 4 */
out(REF REF INT b); /* REF DIOV */
out(REF INT := NIL); /* NIL */
out(DEREF a); /* 4 */
out(DEREF b); /* REF INT */
out({INT a := 3; INT b}); /* 3 a;
   {VOID} b */
out({INT a; BOOL a}); /* {BOOL a} */
out({INT a; a b}); /* {INT a;
   INT b} */

Thus for pointers and environments the first level of indirection is output. The next level is indicated by DIOV and { VOID } respectively. These levels have to be output explicitly. Hidden names are not shown when an environment is output, as the second last example shows. Aliases are also dissolved in the output as the last example shows. Limited denotable values cannot be output. Note that the output is on the form of a correct Nel expression, i.e. it can be used as input later.

Output of a continuation is special. Continuations are similar to the call stack of the program, and could be displayed as a condensed history of the program execution. The items included could be the expressions evaluated to call functions and continuations, i.e. E[ ] [ ] [ ] E, and the offending expression if an error occurred. The line number of the offending expression in a pretty printed version of the enclosing function could be given with the expression. A simple example with the fac function defined in section 2.5.2:

IF MYPID = (FORK twin) THEN
  fac(100);
  STOP
ELSE
  OUTPUT(stdout)(twin); /* {VOID cont;
    VIOD dval;
    VIOD env;
    35 pid;
    {VOID} std;
    REF DIOV status;
    FUN -> NUF pid} */
  HALT(twin);
  OUTPUT(stdout)(twin.cont); /* fac(100);
    fac(n-1); /* 5 line */
    fac(n-1); /* 5 line */
    .... */
  RESUME(twin)
FI

The textual representation is not very enlightening, but as everything can be a function, and everything can be passed as parameters, it is probably the best
we can do. The explicit treatment of continuations is related to debugging of programs, where the different stacks are available, and a more explicit treatment of continuations as indicated in section 5.4 might be desireable.

2.6.2 Character io

There are also functions to input and output single characters, i.e. INCHAR(E1) and OUTCHAR(E1)(E). The relation between OUTCHAR and OUTPUT can be seen from the following example:

```
OUTPUT(stdout)('a');           /* 'a' */
OUTCHAR(stdout)('a');           /* a */
```

2.6.3 External devices

External input devices are like unknown processes inputting to specified denotable values called device variables. External output devices are like processes which echo what is output on a device variable to some external medium.

Device variables are either input or output. The value of the output device variable contains all the text which is written to the device (if it is not read back again :-). We can also output to an inputting device.

To write something to the printer:

```
OUTPUT(printer)(2+3);           /* 5 */
```

This is thus the standard way to get values out of the Nel system, i.e. to an external storage device. Note that a terminal is two devices, one for input and one for output, e.g. tty12in and tty12out.

Devices are accessed through the ordinary environment. This means that we have the recurrent problem of losing track of things when the environment changes, i.e. in an empty environment nothing can be written as no device can be accessed. To alleviate this a bit, three device variables are included in MYPID.std, i.e. in, out and err. Error messages will always be written on err, i.e. SUSPEND will output status on err. We can also use in or out by default by passing VOID as the first argument to input or output. The following two lines are the same:

```
OUTPUT()(4+5);
OUTPUT(DEREFILE(MYPID.std.in))(4+5);
```

Device variables are just a special abstracted form of ordinary memory mapped devices. Other kind of interaction between Nel and external devices can also be defined by mapping the relevant interface of the device into some denotable value. E.g. a bitmap display could be a multiple denotable value of CHAR (8 bits, or BOOL
if we prefer 1 bit), and by assigning to these we could change the bitmap. Any kind of interrupt driven interaction can be accomplished by assigning an interrupt function (an ordinary Nile function) to the denotable value which is called by the interrupt. Synchronization between this interrupt function (the producer) and the device driver (the consumer) is accomplished by semaphores.
Chapter 3

Examples

This section includes some examples of Nel programs, both to get a better feeling of the language, and to relate it to constructs in other languages. Some discussion on the rationales for the design decisions is also included.

3.1 Data types

Functions constructing values

Variables are allocated by evaluating location generating constructs, e.g. INT, 4, FUN ... NUF. When we need multiple copies of variables with the same structure, is it inconvenient to declare the copy of them each time. We can instead make a function which allocates and returns the structure, e.g.:

\{INT a; DIOV b\} r1; \{INT a; DIOV b\} r2;

FUN \{INT a; DIOV b\} NUF tr;
tr() r1;
tr() r2;

array(2,tr) r1:r2;

tr() r1 r2;

The three first pieces of code give the same result (array is defined in section 2.5.2). Note that the last alternative only makes one structure, for which there are two aliases.

This is similar to the use of types as template for variables in languages like Pascal. Note that since Nel has structural equality between values the name equality has to be emulated. For environments this is easy, as we only include an extra declaration to a unique value, e.g.
BEGIN

FUN -> NUF unique;
FUN {unique uniquetype;
   /* rest of environment */
}

NUF
END uniquenv;

If we want to have unique types of non environment values we have to include them in an environment.

Quote in Lisp

Quoting something in Lisp means making a unique constant out of it, which can also be evaluated later by eval. This is modelled in Nel by the following construct:

FUN -> .E NUF name;       /* (define name 'E) */
FUN -> .E NUF name1;
name name2;
name(THISENV);            /* (eval name) */
name = name1;             /* false */
name = name2;             /* true */

The quoted expression is dynamically bound, and all quoted expressions are different if they are not aliases.

In Scheme I can not find eval, which seems substituted by delay and force which gives a call-by-need evaluation of a (statically scoped) expression. These are also easily modelled in Nel:

DIOV promise := FUN E res; promise := FUN res NUF; res NUF;
   /* (define promise (delay E)) */
promise()             /* (force promise) */

Pascal like sets

Pascal like sets (enumerations) are emulated by the UNION constructor in Nel. The components of a Nel union do not have any internal order, so if we want to have an internal order, or use an element of the set as index into a multiple value, we could use the following scheme:

FUN -> up;
   INT i := 1;
   VOID >>
   WHILE i <= up DO
      -> . (VAL i j: i:=i+1; FUN j NUF)
3.2 Constructed operations

Slices

To get a slice of a multiple value, the following function can be constructed:

```c
FUN -> min:max;
FUN -> mv;
    IF (min <= max) AND ((LEN mv) >= max) AND (min > 0) THEN
        INT i := min;
        mv[min] >>
        WHILE i < max DO
            ->, mv[i:=i+1]
        OD
    ELSE
        SUSPEND(/\ REF(min, max, REF mv). THISENV.
        REF ("wrong length for slice"))
    FI
NUF
NUF slice;

slice(2,3)("abcde");  /* "bc" */

slice(monday(),friday())(week) weekdays; /* week from above */

FUN -> start;
FUN -> mv;
    slice(start, LEN mv)(mv)
NUF
NUF rest;

slice(2,(LEN "abcde")("abcde"); /* rest of multiple value */
rest(2)("abcde"); /* rest of multiple value */
```

Note that `slice` only works on the denotable value level, i.e. it does not make any new denotable values, just new aliases.
Variable length operations

Different length assignment is easily implemented from the same length assignment:

\[
\begin{align*}
\text{FUN} & \rightarrow \text{source} \\
\text{FUN} & \rightarrow \text{dest} \\
& \text{IF } \text{LEN(source)} \leq \text{LEN(dest)} \text{ THEN} \\
& \quad \text{slice(dest}, 1, \text{LEN(source)):=source} \\
& \text{ELSE} \\
& \quad \text{SUSPEND}// \text{REF(REF source, REF dest), THISENV,} \\
& \quad \text{REF ("LEN(source) > LEN(dest) in varass")} \\
& \quad \text{FI} \\
& \text{NUF} \\
& \text{NUF varass;} \\
\end{align*}
\]

Note that \text{varass} has to be a higher order function to distinguish the two multiple values, \text{varass (source, dest)} would not work. An advantage of only allowing assignments of multiple values of the same length is that that will give an automatic check in the number of parameters to a function. Default parameters can be modelled by this \text{varass} function. Note the freedom in all kinds of parameter passing schemes with the FUN NUF syntax.

Multiple identifiers

Multiple identifiers are as mentioned in section 2.2.3 just convenient syntactic sugar, i.e.

\[
\begin{align*}
E & \text{ I1:I2;} \\
E & \rightarrow \text{ temp; temp[1]} \text{ I1; rest(2,temp) I2;} \\
\end{align*}
\]

Note that no new copies will be made, all manipulation is done on the alias level.

Character to ASCII conversion

Two simple procedures to convert between the numerical ASCII representation of characters and the characters could be:

\[
\begin{align*}
\text{FUN} & \rightarrow \text{num;} \\
& \quad (/* \ldots 'a','b','c' \ldots \text{ a multiple value */)[num] \\
& \text{NUF atoc;} \\

\text{FUN} & \rightarrow \text{ch;} \\
& \quad (/* \text{ the ordered ASCII set */}) \text{ ascii;} \\
& \quad \text{INT i:=1;} \\
\end{align*}
\]
WHILE ascii[i] <> ch DO
    i := i + 1
OD
NUF ctoa;

Another more fancy way of doing the same is by unification:

FUN -> a:c;
    /* ... REF ('a',41) UNION REF ('b',42) UNION ... */ ascii;
    DEREF(ascii ~ REF(a,c))
    NUF ac;

    ac('a',INT)[2] /* returns 41 */

Note that unification does not change the value of INT, but makes a new value
which is the glb of ascii and REF('a',INT), i.e. REF('a',41).

The comparison on characters can now be defined as:

FUN -> a:b;
    ctoa(a) < ctoa(b)
    NUF chargt;

Equality

 Nel provides two forms of equality, deep value equality = and location equality ==.
To find out if two pointers point to the same value, or two variables contain the
same environment, we have to use the location equality after having passed one
level of indirection, e.g.:

INT a; REF a p1; REF a p2;
p1 == p2; /* false */
DEREF p1 == DEREF p2; /* true */
{INT a} x; DIOV e1 := x; DIOV e2 := x;
e1 == e2; /* false */
e1.a == e2.a; /* true */

Note that location equality does not check for CONST and LIMIT aliases.

3.3 Flow of control

This section is mainly a overview of how continuations are used to emulate different
jump constructs.
Breaks

Breaks are mostly used to jump out of a block of code, e.g. a loop. In Ada exit can terminate a named loop, or if no name is specified, the innermost loop. In C break will terminate the enclosing switch, for, do or while. These constructs are modelled as follows in Nel:

```plaintext
while (exp)
  { /* do something */
    break;
    /* something more */
  }
```

```plaintext
FUN -> break;
  WHILE exp
    /* do something */
    break;()/*
    /* something more */
  OD
  NUF(/)

LOOPNAME:
  loop
    /* do something */
    exit LOOPNAME when FOUND;
    /* something else */
  end loop LOOPNAME;
```

```plaintext
FUN -> LOOPNAME;
  WHILE
    /* do something */
    IF FOUND THEN LOOPNAME;() FI;
    /* something else */
  OD
  NUF(/)
```

The idea is to delay the block until we get hold of the right continuation. Note that we can pass arbitrary values to a Nel continuation, here we only pass VOID.

Labels and goto

The label examples in section 2.5.3 only had backward jumps, i.e. the label referred to already passed code. Forward references are emulated in Nel by enclosing all the code before the label (from the declaration) in a function, to which the label is passed as a value, e.g.

```plaintext
FUN -> a;
  /* do something */
  a;()a;
  /* do something */
  a;()a;
  /* do something */
a;()/* do something */
```

Note that all these label constructs can be evaluated during compile time by evaluating the function which has a constant parameter.

The concept of virtual labels in Simula is actually a restricted form of label assignment, i.e. a label variable is allocated in the superclass, and is assigned by the implicit initialization of each subclass which declares the label. This avoids the
problems of label variables escaping out of the scope of the class. Virtual pro-
ceedures are similarly a restricted form of procedure variables. The emulation of
these constructs in Nel is not to difficult.

Extending continuations

Viewing functions as continuation extenders, as in Filinski’s symmetric lambda
calculus [FILINSKY89], \((c \downarrow f)\) can be modelled by:

```plaintext
FUN -> r: c: f;
    r//(//)(-> p; c//((f(p)))
NUF extcont;

extcont//(c, f) cexf       /* c extended with f */
```

The construction is rather complex, but can be roughly explained as follows:
`extcont` takes the current continuation \(r\), a continuation \(c\) and a function \(f\) as
parameters. It calls the current continuations (returns) \((r//)\) with a new contin-
uation as a result \((-> p; c//((f(p)))\). This continuation takes a parameter \(p\),
applies \(f\) to \(p\) and pass the result to the continuation \(c\).

The `STOP` continuation is made by:

```plaintext
FUN ->//-- // STOP NUF//-- stopcont;
```

I.e. the function is called with the current continuation. This is used inside the
function, and the continuation of the function, i.e. `STOP` is returned, and bound to
`stopcont`.

`STOP` in Nel is similar to Filinski’s `null`.

As mentioned by Filinski, there is a problem of finding a good syntactic (declara-
tive) representation for continuation values. Continuations are not dual to only
values, but values and environments.

The `call/cc` form used in Scheme has an unfortunate consequence which is taken
from the news footer of Griffin:

```plaintext
((call/cc call/cc) 17)
```

which does not seem to be well defined, and is avoided by the `//` construct. The
problem here is that we use `call/cc` as the procedure to which the current contin-
uation is passed as a parameter, and as `call/cc` actually requires two arguments
the result is not well defined.

Exceptions

The exception handlers in Nel are easily extended by extending them with a func-
tion, i.e.
FUN -> reraise:cont:dval:env:status param;
    /* what so ever */
    param /* modified values which are passed on */
    NUF newfun;

TRAP := TRAP &
    {extcont(/ / TRAP.except, newfun) except};

Exceptions in languages like Ada are dynamically bound, i.e. exception handlers are searched for along the call environment. To model this in Nel every function has to take an extra parameter TRAP, which can either be modelled by an extra first parameter or by putting the ordinary function inside a function taking this parameter, i.e:

FUN -> TRAP;
    FUN
    /* ordinary function */
    NUF
    NUF

Iterators

Iterators appeared in CLU [LISKOV77], and were used to iterate through pre-defined and user defined abstract data types, i.e. clusters. These are also possible to simulate in Nel:

    BEGIN
    DIOV next := setiter();
    INT x;
    FOR x IN setiter DO WHILE in(x,next) DO
        OD
    END
    FUN -> x::next;
    (next,x := next//(/(/)) <> (VOID,VOID)
    NUF in;

setiter has to be defined as a function returning a continuation which again returns a value and a continuation producing the next value. A simple example is to iterate through a multiple value:

    FUN -> mv;
    LEN[mv] l; INT i := 0;
    FUN -> ret;
    ret//(/(/);
    WHILE i < x DO
i := i + 1;
ret//(/( // mv[i])
OD
ret//(VOID,VOID)
NUF
NUF mviter;

Iterators are connected to lazy data structures as found in lazy functional languages like Miranda and Haskell [HUDAK88]. The modelling of such structures is also achieved with continuations. Note that infinite lazy datastructures are achieved by always returning a continuation to give the next value.

3.4 Functions

Functions in Nel are totally typeless, i.e. there is no declaration of what kind of parameters and results a given function will take. This is similar to Lisp. Every typing scheme for functions restricts the expressive power of functions in one way or another, or it is undecidable if the given definition satisfies the type. This has led to the decision to make Nel functions typeless. A discussion of some work on the typing of functions is surveyed in appendix D.

Dynamic binding

To get dynamic scoping the current environment is sent as a parameter:

    FUN -> .
    NUF
    /* dynamic scoped body */
    NUF
    NUF dynbind;
    dynbind(THISENV)(/* rest of parameters*/)

Here we see that any attempt to restrict the type of dynbind would restrict the possibility of making a truly dynamically scoped function. Dynamic scoping is convenient when we want to model binding mechanisms on the module level.

Parameter passing

Parameter passing mechanisms are easily modelled in Nel. By using the usual assignment we get call by value. Call by reference is provided by binding the parameters to identifiers.

Call by name is achieved by wrapping the argument in a function, and sending the function as a parameter. Call by text is achieved by making this function dynamically scoped, i.e.
FUN -> arg; arg() NUF fname;
fname(FUN E NUF);        /* call by name */

FUN -> arg; arg(THISENV) NUF ftext;
ftext(FUN ->.E NUF);        /* call by text */

Note that call by text is very similar to how macro preprocessors work.

3.5 Abstract data types

Abstract data types are constructed by functions returning an environment with
access operations to the hidden variables of the adt. Generic abstract data types
are made by sending a type generating function as parameter. The general scheme,
and an example of a generic stack follows below:

FUN -> type;
    /* local declarations */
    { /* interface */ }
    NUF adt;

FUN =: elmentype, size;
array(elmentype,size) stack;
INT top := 0;
FUN -> elem;
    IF top < size THEN
        stack[top:=top+1] := elem
    ELSE
        SUSPEND(//. REF(REF(stack), top, elem),
                THISENV, REF "emptystack");
    FI push;
FUN
    IF top > 0 THEN
        top := top -1;
        stack[top+1]
    ELSE
        SUSPEND(//. REF(REF(stack), top, elem),
                THISENV, REF "fullstack");
    FI
    NUF pop
    {pop pop; push push}
    NUF adtstack;

adtstack(FUN INT NUF,10) intstack;

intstack.push(5);
intstack.pop()
Note the similarity with OWN variables in Algol 60, which are modelled as:

```
BEGIN
  INT a:=0; /* own variable */
  FUN
    a:=a+1 /* body referring to a */
  END funwithown;
```

It is not possible in Nel to specify that the parameter `elemtyp`e should be a type and provide specific operations, like in Ada using the `with` clause. Most violations like those specified in Ada could be detected statically from the code without any further specifications, as the operations will be used in the code. They can also be attempted abstracted to give the interface of the abstract data type, similar to the class abstracter in Eiffel. Determining the constraints of parameters is further discussed in connection with abstract interpretation in chapter 6.

**Multiple interfaces**

As shown in the stack example the function is free to put resources into the returned (exported) environment. The elements of this environment can be put into other environments, and we can restrict the exported components as we want. We can also make the components of environments limited or constant, restricting their use further.

**Private types**

A private type in Ada is a type where only part of the representation is known through the interface. The full representation is only visible for the body. The client can declare instances of a private type, use the public part of the representation, and pass instances as parameters. The interface of the value changes as it passes to the exporting package, e.g. is passed to a function defined in the body. This is mainly a syntactic conversion which is implicitly enforced by the compiler. As Nel is interpreted, and the interface is on the value and not on the type level it is rather complex to emulate this. The only way to enforce this is to register all instances of the type with their limited and full interface, and change the interfaces explicitly by looking up the full one using the limited, i.e.

```
FUN
  DIOV limtypelist;
  FUN -> limval;
    DIOV l := limtypelist;
    WHILE (DEREF l)[1] <> limval DO
      l := (DEREF l)[3]
    OD;
  (DEREF l)[2]
```
NUF lookup;

FUN
    /* create the limited type */
    { /* make the limited interface */ } limintf;
    { /* make the full interface */ } fullintf;
    limtypelist := REF(limintf, fullintf, DIOV := limtypelist);
    limintf
    NUF limtype;

FUN -> limval; lookup(limval) fullvalue;
    /* rest of internal function */
    NUF interfunc;
    NUF adt;

This is a more complicated but also more flexible scheme than the one provided by the syntactic conversion.

Most uses of private types in Ada can be easily avoided in Nel as the use of private types mainly arise because packages are not first class values. E.g. stack variables cannot denote a package, and to hide the representation in Ada the stack package must export a private stack type.

### 3.6 Recursive values

The standard way to make a recursive value in Nel is to declare an alias for a DIOV value which the "recursive" value refers to and is assigned to. This works for functions, references and environments, e.g.

    DIOV fresc := FUN fresc() NUF
    DIOV prec := REF prec;
    DIOV erec := {erect comp};

This scheme uses the indirection between names and values to make the recursive values. The maximal value is still the same (DIOV), and the actual value can thus be changed to what ever.

To make a denotable value with a recursive maximal value the VAL operator is used, e.g. (VAL prec) prect. Note that VAL is able to carry over aliases, thus DEREF prect is prect. Aliases are also carried over by the other structuring operations (unification, copy and type). This also works for functions, i.e. inside the body of (VAL fresc) frect, fresc will refer to the new value, and by changing fresc we do not alter the behaviour of frect. The treatment of the static environment of functions is rather complex and will be explained in connection with unification in section 3.8.

We can still change the contents of a denotable value with a recursive maximal value to something which is a subtype of the recursive value, e.g.:
frect := VOID;
erec := VAL (DIOV a := {a comp; DIOV b});

Note that these subtypes must be either recursive or VOID.

The subtype relation between values with different levels of indirection is rather complex, but roughly speaking a subtype has a level of indirection which is a multiple of the supertype, e.g.

```
VAL (DIOV a := REF REF a) prect := prect;  /* OK */
VAL (DIOV a := {{a comp} comp}) erec := erec;  /* OK */
VAL (DIOV a := REF REF REF a) prect := prect;  /* wrong */
```

The last line is wrong because the level of indirection is 3 and 2, and 2 is not a multiple of 3. For values with non linear indirection, the subtype relation is not so simple and is explained in section 3.8 in connection with unification.

**Rationales for non recursive environments**

Environments in Nel are not recursive, i.e. only the declarations before the current expression are visible. The reasons for this are several:

- Nel is an interactive language, and open recursive environments are not possible in interactive languages.
- Non-recursive environments are semantically simpler.
- Recursive environments can be emulated by passing the total environment as a parameter to the functions.
- Locally recursive values, i.e. functions, references and environments can be constructed by DIOV recv := E. This is similar to forward declarations in Pascal.

### 3.7 Environments and object oriented languages

Environments and the concatenation of such are the basis for inheritance in object oriented languages. This section discusses the environment construct in Nel in relation to object oriented inheritance.

In [CARDELLI84] Cardelli introduced a subtyping concept for unordered records with unique labels. A record, \( r' \) is a subtype of another record \( r \) if \( r' \) contains the same and possibly more fields than \( r \), and the type of the fields of \( r' \) is a subtype of the type of the corresponding fields of \( r \), i.e.

\[
\tau_1 \leq \tau_1 \ldots \tau_n \leq \tau_n \Rightarrow (a_1 : \tau'_1, \ldots, a_{n+m} : \tau'_{n+m}) \leq (a_1 : \tau_1, \ldots, a_n : \tau_n)
\]
"Operations on Records"

In [CARDELLI89] records are as before defined as unordered collections of name value bindings, where all the names are unique. The operations extension, restriction and extraction are defined on these records:

- **Extension**, \(<\ r \mid \text{Name} = \text{Value} >\), extends the record \(r\) with the new binding \(\text{Name} = \text{Value}\). Name must not occur in the original record \(r\). The type of the new record is denoted \(<<\ r.\text{type} \mid \text{Name} = \text{Value.type} >>>\), and it is a subtype of \(r.\text{type}\). The corresponding operation in Nel is \(r \& \{\ \text{Name} = \text{Value}\}\).

- **Restriction** \(r/\text{Name}\) removes \(\text{Name}\) from the record \(r\). \(\text{Name}\) does not need to exist in \(r\). The type of the new record is \(r.\text{type}/x\), which is also a subtype of \(r.\text{type}\). In Nel the restriction is also done with a declaration, i.e. \(r \& \{\ \text{UNDEF Name} \}\).

- **Extraction** \(r.\text{Name}\) gets the value of the attribute \(\text{Name}\) from the record \(r\). Extraction has the same syntax in Nel.

The record types have both positive and negative information, i.e. \(<<</y\) is not equal to but a subtype of \(<<<\), as \(<<</y\) is the supertype of all records not containing a name \(y\). In Nel this is different as \(\{\ \text{UNDEF y} \}\) has no effect on the maximal value of the environment.

Cardelli also introduces a concept of two records being behaviourally equal with respect to a common supertype, i.e. \(<b=2,\ c=1,\ x=3>\) is equal to \(<b=2,\ c=2,\ y=4>\) with respect to the type \(<<b:\text{int}>>\), but not \(<<b:\text{int},\ c:\text{int}>>\). In Nel this corresponds to unifying the two records with the type, and then unifying the results, i.e.

\[
\text{VOIDED}((\{2\; b;\; 2\; c;\; 4\; y\} \sim \{\text{int}\; b\}) \sim ((\{2\; b;\; 1\; c;\; 3\; x\} \sim \{\text{int}\; b\}))
\]

**Discussion**

The record concept here is different from the environments in Nel, which are non recursive and allow redeclaration of names. The principal advantage of having unique names is that:

- Redeclaration and dynamic binding does not allow static type checking. In Simula redeclaration is allowed, but static type checking is assured by static binding. Smalltalk has both redeclaration and dynamic binding, and we can never know which method will be invoked when we send a message to an object.
Modelling of object oriented languages

The modelling of object oriented languages with Nel is not too complex. An example of how the class hierarchy in Smalltalk could be modelled in Nel was given in [KARLSSON88].

The modelling of a static language like Simula is complicated by the following:

- Recursive environments which requires the forward declarations of all functions and procedures as explained in section 3.6.
- The static qualification of references, which can be modelled by making each subclass a distinct environment within the object, i.e.

```plaintext
{{/* declarations only for a */} aqua;
{/* declarations for a and b */} bqua} /* an b object */
refb.bqua.x /* accessing a b object */
```

This means that the qualification, i.e. QUA, must always be explicit, which is naturally as it usually is put in by the compiler.

- INNER which makes the initialization code of the subobject callable from the superobject. This can be generally modelled by making the initialization a procedure which is redefined by each subclass, i.e.

```plaintext
{DIDV INNER := FUN -> NUF;
a.INNER := FUN /* initialization in b */ INNER NUF;
} b;
```

- The coroutine aspect of Simula classes requires a continuation to be passed back along with the environment. This can be modelled by having a variable RESUME which is assigned the current continuation when the object makes DETACH. The modelling of coroutines was also discussed in section 2.5.3.

The actual mapping of the Simula class structure into Nel is not given in detail here as it is quite complex, but a similar mapping for Prolog is given in appendix E.

### 3.8 Unification

Unification in Nel is a quite complex operation. It extends the ordinary Prolog unification in the following ways:

- It is based on sets of values, and not only having the distinction variable/ground. (Here a partially instanciated data structure is considered ground, as the top level can not be further bound.) This is similar to the unification presented in [AIT–KACI86] and [HUBER87].
• Unification is well defined for recursive structures, not only for trees. This
  is similar to work presented in [COLMERAUER82].

• Unification is defined for functions (and continuations\(^1\)). This is quite simple
  except for the static environments, as all functions (and continuations) are
  unique, and can only be unified with itself or DIOV. In the latter case a copy is
  created where the environment of the copy is updated to refer to the unified
  denotable values.

Simple examples

Unification works both on the maximal and actual value, e.g.:

\[
\text{(INT}\ a := 3) \sim\ \text{DIOV}\ (b := \text{FALSE})\ /\!\!*/\ \text{INT} := \text{VOID}\ */
\]

Unification of a supertype with a subtype will give the subtype, e.g.:

\[
\text{INT} \sim 3\ /\!\!*/3*/
\]

Unification of multiple values unifies each component, and they must have the
same length, e.g.:

\[
\text{INT,DIOV} \sim \text{DIOV,TRUE}\ /\!\!*/\ \text{INT,TRUE}*/
\]

Function components

Unification of structures with functional components require some extra care. A
function unified with DIOV should give a similar function. The problem is the static
environment of the function.

Consider the following example:

\[
\{\text{INT}\ a;\ \text{INT}\ b;\ \text{DIOV}\ c := \text{FUN}\ a+b\ \text{NUF}\}\ \times;
\{\text{INT}\ a;\ a\ b;\ \text{DIOV}\ c := x.c\}\ \times;
\times \sim y\ /\!\!*/\ \{\text{INT}\ a;\ a\ b;\ \text{DIOV}\ c := \text{FUN}\ a+b\ \text{NUF}\}*/
\]

The resulting function should refer to the new a and b which are aliases.

This works as follows for COPY, VAL, TYPE and unification. All of the components
of the original structures which are traversed are marked, and linked to the result.
Functional components in the new structure are marked, and their static environ-
ments are resolved when the operation is finished on the other values. This is done
by traversing the old static environment, and when it refers to a component which

\(^1\)The correct treatment of continuations requires the explicit treatment of continuations in the
semantic model which is indicated in section 5.4.
is marked, the new static environment is made to refer to the result. Continuations are treated similarly to functions. The details of the algorithms are given in chapter 5.

Note that the components of the implicit environment will not be copied, i.e.

\[
\text{COPY \{THISENV a; a.FUN \ldots NUF x\} e1;}
\text{COPY \{FUN \ldots NUF x\} e2;}
\]

are different. \(e1.x\) will use a copy of the total environment which is created by COPY, whereas \(e2.x\) will use the old environment. VAL, TYPE and unification work similarly. Thus extensive data structures like THISENV should be avoided as arguments to these functions if they are not intended, i.e. making a backup copy of the global environment.

**Aliases**

Unification of aliases work similar to copying, i.e. there will only be made one copy of the aliased variables. For example:

\[
\{\text{INT a; a b := 3} \} \sim \{\text{INT a; INT b}\} /* \{\text{INT a:=3; a b}\} */
\{\text{INT a; a b} \} \sim \{\text{INT a; 3 b}\} /* \{3 a; a b\} */
\{\text{INT a; a b} \} \sim \{4 a; 3 b\} /* \{VOID a; a b\} */
\]

\(\text{(DIV \ a, \ a, \ DIV \ a)} \ c1;\)
\(\text{(DIV, \ DIV \ b, \ b)} \ c2;\)
\(a \ c3;\)
\(c1 \sim \ c2 /* (DIV \ a, \ a, \ a) */\)

These examples had only internal aliases in each structure. Aliases between structures which are unified are treated by making one copy. Some examples:

\[
\text{INT a;}
\{\text{INT x} \} \ b;
\{\text{b.x x} \} \ c;\]
\(a \sim a /* \text{INT */;\)}
\(b \sim b /* \{\text{INT x}\} */;\)
\(b \sim c /* \{\text{INT x}\} */;\)

This means that aliases between the arguments are treated as external aliases, and will not appear in the result. If we want to also include aliases between the structures \(x\) and \(y\) in the unification the following scheme can be used:

\[
\{\text{DIV \ a; \ a b} \} \sim \{\text{x a; \ y b}\}
\]

The unification of aliases is the mechanism used in unification in logical languages. Note in the last example \(c3\) will still be an alias for the first DIV, as it was not
part of the unification, i.e. only the alias explicitly contained in the unification arguments will appear in the unified value.

This is used in the modelling of Prolog which is given in appendix E, as all variable bindings are passed around. This can easily be explained in terms of substitutions (environments), the unification makes a new substitution using the old and the local; $\theta_n = \theta_o \sim \theta_l$.

Note that the unification process is complicated, i.e. it requires multiple passes over the arguments as the following examples shows:

\[(\text{ DioV} \ a, \text{ DioV} \ a) \sim (\text{ DioV} \ b, \text{ DioV} \ b) /* (\text{ DioV} \ c, \text{ DioV} \ c, \text{ DioV} \ c) */ \]

A one pass solution will prematurely make a copy of the second component, which should actually be an alias for the first.

**Recursive structures**

Unification is also defined on recursive structures, e.g.

\[
\begin{align*}
\text{ DioV} \ a & := \text{ REF REF} \ a; \\
\text{ DioV} \ b & := \text{ REF REF REF} \ b; \\
\text{ a} & \sim \text{ b}; /* \text{ DioV} \ x := \text{ REF} \ x */
\end{align*}
\]

Thus the unification will give a retraction.

\[
\begin{align*}
\text{ FUN \{DioV} \ a; \text{ DioV} \ b\} \text{ NUF envt}; \\
\text{ array(envt.6) x1:x2:x3:y1:y2:y3}; \\
x1.a & := x2; \ x1.b := x3; \\
x2.a & := x1; \ x2.b := x2; \\
x3.a & := x3; \ x3.b := x1; \\
y1.a & := y2; \ y1.b := y3; \\
y2.a & := y2; \ y2.b := y1; \\
y3.a & := y3; \ y3.b := y3; \\
x1 & \sim y1 /* \text{ DioV} \ r := \{\text{ DioV} \ a := r; \text{ DioV} \ b := r\} */
\end{align*}
\]

### 3.9 Concurrent constructs

This section gives some simple standard examples of the use of the concurrent constructs in Nel. Examples of the use of the more operating system like features like e.g. HALT are not given.

**Dining philosophers**

This is a classical example originally due to Dijkstra [DIJKSTRA68].

\[\text{array and subset are examples from 2.5.2}\]
array(FUN SEM NUF, 5) forks;

FUN -> := (subset(1,5) i)
  WHILE TRUE DO
    P(forks[i]); P(forks[i+1 MOD 5]);
    /* eat */
    V(forks[i]); V(forks[i+1 MOD 5]);
    /* think */
  OD
NUF phil;

INT i := 1;
MYPID m;
WHILE i <= 5 DO
  INT j := i;
  IF FORK = m THEN phil(j) ELSE i:=i+1 FI
OD

Non-blocking semaphores

There is also no implicit way of checking the value of a semaphore before doing the P operation. Nonblocking semaphores will have to be modelled explicitly by for instance:

FUN
  SEM s; SEM m; INT num := 1;
{} FUN -> a;
    P(m);
    IF num = 1 THEN
      num := num - 1; V(m); P(s); TRUE
    ELSE
      IF a THEN
        V(m); FALSE
      ELSE
        num := num - 1; V(m); P(s); TRUE
      FI
    FI
NUF nbp;
FUN
  P(m); num := num + 1; V(s); V(m);
  NUF nbv
}
NUF nonblockingsem:
3.10 Software development in Nel

This section contains a short scenario for how software development is supposed to be done in Nel.

Software development consists of structuring, versioning and rebuilding the system.

Structuring the system

A program can of course be written as one monolithic function. This soon becomes messy as the system gets larger. Then this function can be split into smaller functions internally, and the main function calls these smaller functions. The next step is to move these smaller functions out of the main function. Thus the main function will use but not contain the smaller functions. As Nel is statically scoped at least the names of the smaller functions must be defined before the main function. The smaller functions must also be relatively self contained, i.e. not referring to global variables in the main function. The next step is to place some functions in a separate library (environment). This library can either be connected statically to the main function, or provided as a parameter.

The software system will now look like:

```nel
{FUN /* small fun 1 */ NUF f1;
  FUN /* small fun 2 */ NUF f2;
  /* and so on */
} lib;

FUN lib&THISENV.( /* main program statically connected to library */)
  NUF main1;

FUN -> lib;
  lib&THISENV.( /* main program, library provided as parameter */)
  NUF main2;
```

Note that the interface of the program changes drastically from main1 to main2, as we have to provide a library parameter, thus main2 is like a parametrized module. This extra flexibility is only needed if different instances of main2 is intended to use different components. Note also that main2 is still connected to the static environment through THISENV.

In main2 we pass the library to the function when we call (run) it. We could also distinguish between binding and running by yet another function, i.e.:

```nel
FUN -> lib;
```
lib&THISENV.
FUN
/* main program bound to library */
NUF
NUF main3;

Note that changing one of the functions in the library will affect the inner function as it is not looked up before it is used.

The functions in the library could of course also go through the same process as the main function. If some of the functions in the library are parametrized, we will have to provide parameters for them from the main program. The instantiation (binding) can be done once if the function is written as main3, as the resulting function can be bound to a new variable.

Changing a function

When we take a function into the editor, we get a copy which has the same static environment as the original. If the original function was stored in a denotable value with maximal value DIOV, we can overwrite the original function with the modified one. The original is thus lost.

Often we want to record the changes done to a function, so that we can get back to a previous version. In change oriented versioning \(^3\) this is modeled by introducing an option, say opt, which characterizes the change. To get the new version, i.e. the file with the change we set opt:=TRUE, and to get the original version we set opt:=FALSE. The changes will appear as:

FUN -> opt
FUN

IF opt THEN
    /* new text */
ELSE
    /* old text */
FI

NUF
NUF

Note that we have to provide the value of the option as a parameter to a function which returns the desired version. This is very similar to conditional compilation, the only difference is that it is on the language level, i.e. the syntactic instead of the lexical level.

It is not very convenient for the user to look at all this redundant code, and to manually insert all changes. We also have a problem with changing environments,

\(^3\)A short introduction to change oriented versioning is given in appendix C
i.e. \texttt{opt} will not be visible inside \texttt{E2} in \texttt{E1,E2}. The user has to manually export a into \texttt{E2}, i.e. \texttt{(E1&\{opt opt\}) .E2}, if some changes appear inside \texttt{E2}. Another problem is overlapping names in new environments, i.e. any \texttt{opt} in \texttt{E1} will be hidden. In conventional systems these problems are avoided by using the variables controlling the conditional code on another level, i.e. the preprocessor or the versioning system, e.g. SCCS. The real program text is then just viewed as data, and the problem with name clashes does not occur.

To alleviate some of these disadvantages the DEI editor described in section 4.6 has support for the automatic insertion of this extra code to support the versioning. This extra code is also removed before the function is edited.

The version editor works as follows. It expects a function on the form:

\begin{verbatim}
FUN /* version prelude */
  -> Oold &
  \{ Oold choice;
   FUN ->. (o1 AND NOT o3 /* ... */) NUF ambif(choice) amb1;
   FUN /* other ambitions */ NUF ambln(choice) ambn
  \} 0;
FUN /* versioned function */

IF O.ambi THEN /* old changes */
  /* new E */
ELSE
  /* old E */
FI

(E1&(O O)).E2 /* changing of environment, skipped if no changes in E2 */

E(0.choice)(E) /* calling a versioned function */
\end{verbatim}

When the editor is invoked with a versioned function, it searches the calling environment for environment "0" where it expects to find a function "0.amb", an environment "0.choice" and a reference to a multiple text value "0.ambname". "0.choice" in used for "Oold", and the old ambitions are evaluated and simplified. The "0" environment is extended with \{ Oold.amb amb; Oold.ambname ambname \}. The prelude function and the dead code is then removed, and only the chosen version(s) is displayed. If the choice is not complete enough to determine all old ambitions, we will still see some fragments on the form:

\begin{verbatim}
IF 0.ambi THEN /* old changes */
  /* new E */
ELSE
\end{verbatim}
This is called multiversion editing, and corresponds to editing a file with conditional compilation directives. By looking up “0.ambif” in the static environment we will see what options remain to be bound. By changing “choice”, and reevaluate the selection process we can get another view.

Editing on the file will appear as ordinary editing, but as IF expressions in the fully versioned file. A copy of the new ambition is included in the prelude, it is given the name found in the text “0.ambname”, and will appear as the test in the IF expressions.

The advantage of versioning on the syntactic instead of the lexical level is that we can assure that we will get a syntactically (context free) correct program for any choice, i.e. merging. As the syntactic correctness is usually the first step when merging two changes, we have gotten one step ahead.

Note that a versioned function is called by first giving the choice as a parameter, i.e. \( f(0.\text{choice}) */\text{ ord param} */\). If we only want one version we can instantiate it with only the choice, and bind it to a new name. This is similar to functions parametrized over a library or “type” generating function.

Thus versioning, different libraries and genericity (type generating parameters) are all expressed as function application in Nel. This is possible because of the lack of type constraints on the function values.

The multiversion editing in the Nel editor, DEI is further discussed in section 4.6
Chapter 4

The Nel Programming support environment

4.1 Introduction

This section describes the integrated programming support environment for Nel. The Nel environment consists of one tool, DEI. DEI is a combined interpreter, editor and debugger which resembles Emacs. This environment is supposed to replace all the ordinary tools used to construct a software system. DEI replaces the operating system, debugger, tester and editor. Function application in Nel is flexible enough to incorporate selection of versions, composition of components, including binding of libraries and instantiation of generic components. An additional tool, PE (partial evaluator), is discussed in chapter 6. PE should do partial evaluation based on abstract interpretation, combined with caching of results. PE should replace the traditional role of the deriver tools to produce a more efficiently running program.

A Nel invocation

An invocation of the Nel system is called an invocation, and is started with an initial store and environment which typically contains the connections to the invoker. An invocation can be bootstrapped on the bare machine, started from the host operating system, or by a process in an invocation. The command to start an invocation from a Nel process is \texttt{NEL E}, where \texttt{E} should evaluate to an environment.

An invocation starts up DEI, and appears as an Emacs editor to the user. The main differences are that Nel interaction buffers and the minibuffer corresponds to processes, and that statically scoped interaction buffers are associated with visited Nel functions.

It might be helpful to think of the minibuffer as the controlling process, and all the interaction buffers as children.

Commands to DEI are given as commands to Emacs. \texttt{M-x suspend-dei} will sus-
pend this session, and return to the invoker. M-x `save-buffers-store-dei' terminates this session. Note that each user in Nel is typically supposed to work inside one invocation of the editor, so to halt or stop it is quite drastic.

The typical way to set up a multiuser Nel invocation is to let the administrator start the Nel invocation. Then she starts some interaction buffers, changes `std. in', `std. out' and `std. err', starts DEI in each of them. The other users can now connect to these processes.

4.2 Controlling DEI

The minibuffer

The minibuffer works as for Emacs. Each buffer has its own environment. These environments are collected in the `buffers' environment in the minibuffer. Thus C-x C-f from a buffer `xbuff' will give the automatic prefix `buffers.xbuff'. This can as in Emacs be edited. The variable `currentbuffer' denotes the last active interaction buffer before invoking the minibuffer. All expressions given to DEI commands (except M-x `nel') will be evaluated in the environment of `currentbuffer'.

To give a general Nel expression to the minibuffer the M-x `nel' command is used. Any Nel expression can be given to the minibuffer and will be interpreted as input to a clone of the minibuffer process, which the real minibuffer waits for. The real minibuffer can be halted and resumed from the invoking process (or hardware) if the cloned process does not terminate and should be halted. Note that all new declarations will be local to the cloned process. The declarations visible in the minibuffer are those included in the initial environment passed as parameter, and those used internally to record the status of DEI.

All input to DEI is going through the minibuffer process, which displays it in the right buffer, and distributes it to the other processes.

The buffers

The buffers are given names automatically by the editor, and the name is displayed with the contents as for Emacs. The operations on buffers are the same as in Emacs, except the corresponding interaction buffers are removed together with their function buffer.

There are four main kind of buffers

- Interaction buffers, suffix `i'. These are connected to interpretation processes, and the input directed to such a buffer is given to the interpretation process.

- Function buffers, suffix `f'. These are like the textual representation of Nel values, and appear like text files. There are special operation to manipulate
versioned functions, which are described in section 4.6. The denotable value of the function buffer together with the environment in its initial interaction buffer can be used as an ordinary function, i.e. called.

- Environment buffers, suffix ‘e. The textual representation of environments, but with special operations to manipulate them like dired in Emacs.

### 4.3 Interpreting in DEI

Similarly to the Lisp interaction buffer or shells in Emacs there are Nel interaction buffers. These buffers are statically scoped.

Interaction buffers are asynchronous, i.e. we can start something in one buffer, and move to another one doing something else.

The interpreter parses, evaluates and prints the result of Nel expressions. One process executing the interpreter is active in each interaction buffer. The minibuffer also interpretes closed Nel expressions. The input directed to an interaction buffer is sent to the corresponding interpretation process. The input is given to the interpretation process with the command M-x eval-region. If a syntax error is found in the expression none of it will be evaluated, and an error message is output to std.err.

If an evaluation error occurs while interpreting the expression, the interpretation process is suspended, but will be immediately restarted with the old continuation, environment and denotable value which it had before starting to parse this expression, and an error message is output to std.err. The error environment in the minibuffer keeps the dval, env and cont of the latest error for each interaction buffer.

The interpretation can be interrupted by M-x keyboard-quit from the buffer. The interpretation process can also be halted from another process, e.g. the minibuffer, with HALT(E).

An example:

```plaintext
BEGIN
INT x;
INT b; x := 3; c
    /* undefid c appears on std.err */
OUTPUT(x)
    /* 3 appears on std.out, i.e. x has changed */
b
    /* undefid b appears on std.err, 
      the environment does not include b */
NUF
    /* NUF found END expected appears on std.err */
```

Note that the interpreter parses [ ; | >> ] E [ ; | >> ] as correct expressions.
A closed expression need not be entered as one expression.

Expressions evaluated in a Nel interaction buffer have the same effect as evaluated in a stand alone program.

One interesting application of explicit continuations is that saving the current continuation can transfer "work contexts" between users.

**Forking**

The input to the interpretation process is passed from the minibuffer process through a text variable, thus forking the interpretation process is not recommendable. To make a clone of the interpretation process which is connected to a new interaction buffer M-x `fork-Interaction-buffer' is used.

**Halting and stopping**

When an interpretation process is halted, it will automatically be restarted by the DEI, but DEI is only concerned about the initial process, thus any clones can be halted and restarted without interference from DEI. The immediate restarting of a halted interpretation process can be switched off. The explicitly restart the interpretation process M-x `restart-process' is used. A stopped interpretation process will not be restarted. To delete the interaction buffer M-x `kill-buffer' is used.

The status of the interpretation process behind the buffer is displayed in the status line. These are:

- **ready** waiting for input
- **running** processing input
- **halted** by an error or by some other process.

**Input and output**

Each interpretation process' `std.in', `std.out' and `std.err' are initially connected to its interpretation buffer, or rather the input and output denotable values associated with the buffer. Thus output from the process will be written to out, and eventually displayed in the buffer by DEI. Input will be read from in, which is the same place as the interpreter takes its input from.

If the input and output are separated in different buffers (windows) they can later be used for testing, i.e. save the manually given input, and the corresponding output, and rerun the function with the input and output redirected, checking the new results against the old.

Care should be taken so that the interpreter process is connected back to the interaction buffer.
Comments as DEI commands

Comments are scanned by the interpreter, and if they contain a DEI command, the command is executed (given to the minibuffer). Only one command can be given in each comment, and they are on the form of written Emacs commands, e.g. /* M-x nel E */ where E is an expression, and /* C-x C-s */ or the equivalent /* M-x save-buffer */.

4.4 Visiting a function

Getting a copy of a function into the editor is done by M-x find-dval name (C-x C-f name). Limited variables can not be visited (read), and constant functions can not be overwritten. This is called to visit a function. The buffer containing the copy will be named as in Emacs. If the name is not unique, DEI will give the buffer a default name, i.e. funcx'f.

Interaction buffers

Together with each function there is an interaction buffer where the static environment of the function is available. Changes done to the static environment (new declarations) will only have effect on the static environment of the copy of the function while it is visited. It is mainly useful for making dummy static variables, thus not affecting the original variables during tracing. A function inserted (M-x insert-dval (C-x i)) into a buffer will lose its static environment. The static interaction buffer is called funcname'i.

Executing a visited function

If a visited function is called from an interaction buffer the original value in the store will be fetched and executed. To use the visited copy the buffer name must be used, i.e. buffers.funcname'f. The copy is only available through the minibuffer, but the denotable value can be moved to any interaction buffer, e.g:

    DIOV tempfunc;    /* in tobuffer   */
    buffers.(tobutton'i.tempfunc:=REF(funcname'f)) /*in minibuffer*/

This is the standard way to move values between the different environments.

When a function buffer is executed a copy of the text and environment is taken, and the piece of text is copied when it is traced.

Creating a function buffer

The command M-x function-buffer name will make a new empty function buffer with static environment name. An interaction buffer is also created.
It is thus similar to name.FUN ... NUF.

4.5 Tracing in DEI

A visited function can be traced in DEI. This is done by \texttt{M-x trace buffer parameters}. The parameters are evaluated in the environment of \texttt{current buffer}. A new interaction buffer with the static environment of the function is then created. Tracing works similarly to entering expressions of the function manually in the interaction buffer.

While tracing a function in a buffer a mark will indicate how far the tracing has proceeded. The tracing can any time be interrupted with \texttt{C-g}. This will also stop functions which are executed in one batch. Tracing will also be interrupted by an error. The process identifier in \texttt{error} can then be used to continue the tracing after some fixes.

Tracing proceeds by default to the next \texttt{>>} or \texttt{;} by \texttt{M-x trace-expression}, requiring the user to resume the tracing for each new expression. The resume command can also be given a repeat argument. \texttt{M-x trace-to-point} and \texttt{M-x trace-to-text text} are other functions. There is also a slow motion switch. The mark indicating how far the tracing has proceeded can also be moved.

When the traced function returns (\texttt{NUF}) the interaction buffer is deleted. The result of the traced function will be returned to the buffer invoking the trace.

\textbf{Interaction buffers}

A function can be traced through many interaction buffers. If we want to save the interaction buffer we only have to fork that buffer. These buffers have to be killed manually when tracing is finished.

We can also store the environment when the tracing is interrupted in another buffer via the minibuffer.

The marks in the traced buffer are retained for all interaction buffers used for tracing the function. It can be stored as a mark variable in connection with the interaction buffer.

\textbf{Tracing of nested functions}

There are problems in integrating tracing and editing in a language where functions are first class values, and created dynamically. The problem is that we do not know where the original text which created the function is.

Consider the following example:

\begin{verbatim}
FUN
\end{verbatim}
FUN
    ...
    NUF x;
    ...
    x(...);
    NUF b;

Visiting, tracing and editing $b$ works fine. The problem is $x$. If we evaluate the
declaration, a "copy" of $x$ will be stored. Stopping the tracing inside $b$, and reading
$x$ into the editor, we can trace and edit it, but the "original" $x$ which is created by
$b$ each time it is run is not affected. This is not a great problem, as we can replace
$x$ in $b$ when we are satisfied with its behaviour.

Some functions creates multiple copies of functions, e.g. abstract data types, where
each copy has a different static environment (example in section 3.5). Changing
one of these copies will not affect the others, and changing the generator function
will only affect future copies. Most of these cases should be collapsed into one
piece of code with changing environment by partial evaluation, but that does not
change the conceptual view.

If we know we want to use a copy of the function in a buffer we can redefine the
name of the function to refer to the buffer instead, i.e. insert the following just
after the declaration of $x$.

    /* C-x C-f x */
    BEGIN
        DIOV temp;
        /* M-x nel currentbuffer.temp := REF buffers.x'f */
        DEREF temp
    END x;

Any call to $x$ will now use the copy in the buffer. To trace every call we redefine
$x$ once more:

    CONST FUN -> p
    /* M-x trace x p */
    NUF x;

Thus any call $x(E)$ will invoke a tracing of $x$. We make $x$ constant to avoid
overwriting the buffer.

Recording interleavings

When tracing parallel processes the user usually interleaves them manually with
different trace commands. This interleaving information will consist of a list of
trace commands, and are recorded as expanded Nel expressions together with all
other commands in the minibuffer. By saving these commands as text, we can
replay the interleaving.
Editing a traced function

Traced functions can be edited just like ordinary text, and the new code will be used when tracing.

Watchpoints

Watchpoints are different kind of checks on variables, i.e. interrupt when read, written or when reaching a given value. By redeclaring variables in the static environment of the visited function we can achieve these effects, i.e. redeclaring a variable as constant will interrupt each time it is written, making a limited variable will interrupt each time it is read. By making an alias for the original variable we can manually do the correct assignment afterwards. By excluding the values we watch out for from the maximal value of a variable we can get an interrupt when an attempt is made to assign any of these values to it. More complicated watchpoints have to be implemented as functions. These functions must then be "manually" invoked to check the condition. By including calls on the watchpoint functions together with tracing commands in a function, we can get them checked each time we resume the tracing. The watchpoint functions can also be included anywhere in the code, either as comments and executed by the minibuffer, or as ordinary code.

\begin{verbatim}
M-x nel buffers.watchbuf\'i.watch := (currentbuffer.FUN ... NUF)
M-x nel buffers.watchbuf\'i.watch()
\end{verbatim}

By putting the watch function into a separate interaction buffer we are sure that it can be invoked through changing environments in the traced function.

4.6 Versioning in DEI

The basic support for versions of a Nel function was described in section 3.10. This section gives more detailed description how DEI support change oriented versioning.

Looking up ambition and choice

The variables \texttt{0.amb} and \texttt{0.choice} are looked up in the environment from which the function was visited. These values are copied, and associated with the visiting buffer. If they are not found, and the function is versioned, DEI will visit the "raw" function, but give a warning.
Internal representation of changes

A versioned program can be pictured as a directed acyclic graph (dag).

Take the following generic example: \( \text{E1 \ op \ E2} \) is changed to \( \text{E1 \ op' \ E2'} \), which will internally be represented as:

![Diagram of internal representation of changes]

Figure 4.1: Internal representation of changes

Common fragments in \( \text{E2} \) and \( \text{E2'} \) will also be recorded, just as the common fragment \( \text{E1} \) of the whole expression was recorded.

A note on terminology might be in order. A node is a token in the syntax. A fragment is a list of textually connected nodes.

Detecting common fragments

When a versioned function is read into DEI the versioned part is parsed, and each node is given an unique identifier. This unique identifier follows the node in the buffer, even when it is moved. But when a node is copied it loses its identifier. Nodes in common fragments are given identifiers once.

Changing the function text consists of inserting new nodes, deleting old ones, and moving nodes around. Changing a node (token) amounts to deleting the old, and inserting a new. Copying nodes is treated as inserting new nodes.

This is different from the approach taken in versioning objects with real identity where the visibility is something connected to each object, and has to be updated to \( v \land \text{amb} \), and \( v \land \neg\text{amb} \) for the changed and the original fragment. The visibility here is a property inherited down the syntax tree. When we move a fragment from one place to another the new total visibilities will be \( v_2 \land \text{amb} \), and \( v_1 \land \neg\text{amb} \), where \( v_1 \) is the inherent visibility on the original place, and \( v_2 \) is the inherited visibility on the new place. For common fragments the total visibility is the disjunction of all the paths leading to this fragment.

When the changes are done, and the new function works the changes are ready to be integrated in the multiversion function with the specified ambition.
The integration is done as follows (we suppose that the node identifier is a pointer into the original syntax tree):

- Expand the changed function with the hidden parts.
- Traverse the syntax dags of the changed and the original function. There are three cases:
  1. The node identifiers are the same. Proceed with the subtrees of both nodes.

```
    id
   /\  
  /   \
 id   id
    \  /
     \id
      Original
```

```
    id
   /\  
  /   \
 id   id
    \  /
     \id
      Edited
```

```
    id
   /\  
  /   \
 id   id
    \  /
     \id
      Merged
```

**Figure 4.2: No changes to the node**

2. The node in the changed function is new. This means that something is inserted. Make a new \textbf{IF} node in the original tree. The ambition is the test, the new node is the THEN branch and the old is the ELSE branch. Proceed with the subtrees of the changed function. As long as new nodes are encountered copy them to the THEN branch. When a node with an identifier is encountered again, go to case 1.

```
    id
   /\  
  /   \
 id   new
    \  /
     \id
      Original
```

```
    new
   /\  
  /   \
 id   id
    \  /
     \id
      Edited
```

```
    ID
   /\  
  /   \
 id   amb
    \  /
     \new
      Merged
```

**Figure 4.3: Inserted nodes**

3. The node (\textit{id'} ) in the changed tree is not the one expected (\textit{id}). This means that something is moved or deleted. Make a new \textbf{IF} node with ambition as the test. Let the THEN branch be the node, \textit{id''}, in the original tree corresponding to \textit{id'}, and let the ELSE branch be \textit{id}. Proceed by comparing the subtrees of \textit{id'} and \textit{id''}.

- The merged syntax dag will now contain the merged function.
Figure 4.4: Moved or deleted nodes

Note that deleting a token and inserting the same token again is not the same as doing nothing. The difference is that the algorithm above does not look at the value of the token, just its identifier. Deleting and inserting ; in E1;E2 is illustrated in figure 4.5.

Figure 4.5: Identical changes

The difference is that changing ; to >> will only affect the visible branch. To avoid possibly unintentional changes like these DEI can detect and display them M-x display-common-nodes. M-x collapse-nodes will connect them together. Note that E1 and E2 need not be the same in both branches for the top node to be removed.

Displaying a single version

A single version is displayed as an ordinary program. Removed nodes can be optionally displayed with parenthesis in comments, i.e. /*(/ /*)/. Thus the dubious example from above can be displayed as:

E1/*(/;/*/)*/E2

Selecting either of these comments DEI will display the alternative tokens and the ambition in a read only buffer, e.g.:
NOT amb -> ;

Note that if the node is in the positive branch, the ambition will be shown preceded by NOT. The buffer show the latest committed information, but will not update this information automatically.

The displaying of versioned text can obviously be greatly improved by replacing the textual indicators presented here with colours or fonts on a bitmapped screen.

**Multi version editing**

When a choice is not complete enough to determine an ambition (a visibility), DEI will display the multiple version. Common fragments are only displayed once, by default at the place they occurred first in the text. The number of the (textually) first node of the common fragment is inserted around the displayed fragment. The other occurrences of the fragment are shown with the number in a comment. The same dubious example:

```
IF amb THEN
   /*(223*/ E1 /*223)*/ ; /*(317*/ E2 /*317)*/
ELSE
   /* 223 */ ; /* 317 */
FI
```

Showing the number around the displayed fragment can be turned off. There is also a command to change where the fragment is displayed, e.g. removing the numbers and changing the display of the fragments to the else branch will give:

```
IF amb THEN
   /* 223 */ ; /* 317 */
ELSE
   E1 ; E2
FI
```

Editing a common fragment will, as when the other fragments were hidden, result in the changing of all the other fragments too. The hidden fragment can also be copied to a read only buffer, but as for showing hidden fragment in a single version, this buffer will not be kept up to date with later changes.

A fragment consisting of many tokens can be split so that part of it is shown in different buffers.

To disconnect a fragment from its common group a copy is taken of the fragment, the original fragment is deleted, and the copy is inserted. When a common fragment is deleted at the place it was displayed, it will be displayed somewhere else. Any changes done to the copied fragment will not be propagated.

Note that E1 and E2 from the above example can also contain common fragments.
Moving fragments between denotable values

Restructuring a software system is always a traumatic experience (and should be avoided :-). It consists of shuffling pieces of code around between different denotable values (functions in Nel, separate compilation units in Ada). Section 3.10 contains a small example. The problem occurs when we change the moved or original code, and we want the other also to be updated. The standard way of doing this is by factoring the common parts into files for textual inclusion.

To automatically support this DEI will not move the code into the other denotable value, but create a shadow copy which is transparently referred from both the old and the new denotable value. Visiting one of these functions will include the shadow copy in the text, with a proper node name. If the other function is also visited it will only display a comment where that common subexpression is, e.g. /* buffer::223 */. This will of course only occur in multi version editing, as otherwise one of the fragments are hidden. The buffer to display the fragment can be chosen as for fragments within a buffer. Note that the shadow copy is not directly accessible for the user. A reference count is also included in the shadow copies, and the shadows are included in the remaining value when it becomes one. When we copy a value with shadow fragments to another denotable value, the shadow fragments are first merged into the value, and the total value is copied, thus the connection is lost.

Restructuring a software system

The above paragraph only dealt with reorganization within the given denotable values (the environment remains the same). To reorganize the environment structure (module structure) is even more traumatic. And if we want to have some form of connection between the old and the new version, we cannot get much support from current systems.

In DEI the reorganization of the environment structure can be achieved in two ways, either by making an extended environment structure incorporating both the old and the new, or by versioning the environments.

The old environment is a subenvironment of the new:

- Make a temporary environment, which includes a copy of the old, let the actual value of all the new functions be empty, i.e. FUN -> NUF, and all the old be D0IV.
- Unify the old environment with the temporary, giving the new, i.e. old ~ temporary new.
- Test it to see that the new part of the environment does not interfere with the old.
- Overwrite the old and the temporary environment.
- Reshuffle the product as explained in the previous section.
When the changes are small, this is the preferred approach. A function could possibly be made for doing the first two steps.

The old and the new environments are incompatible:

- Make the new empty environment as above.
- Reshuffle the system by moving it from the old structure to the new with the given ambition.
- Make a function to select the old or the new environment, i.e.

```
FUN -> choice
    IF amb(choice) THEN
        OUTPUT("use new environment")
    ELSE
        OUTPUT("use old environment")
    FI
    FI
NUF envselect;
```

Note that the ambition can overlap both environments, but we have to have a choice which selects one.

**Checking choices**

Every complete and incomplete choice will give a syntactically correct Nel program, thus we can for instance set up a function to explore different choices by selecting them, doing partial evaluation, and running them through test rigs. This can be done as a batch job, only collecting the results. We could also explore larger parts of the choice space by running partial evaluation on an incomplete choice. The partial evaluator should treat the unbound options as unknown, and do the best it can. This might be faster than exploring the choices one by one.

### 4.7 Other values in DEI

**Editing environments**

Just like Emacs has a mode for editing directories, dired, DEI has a mode, EnvEd, for editing environments. A copy of the environment is displayed. The environment can be manipulated as in dired. The extension from dired is to insert declarations, i.e. M-x insert-decl name1 name2, where name1 can refer to any denotable value reachable from the minibuffer.

EnvEd makes a new environment, which can be stored anywhere. The new environment contains aliases for the nondeleted names of the old environment. Functions from the old environment have not changed their static environment.

72
Editing text files

Text files in Nel are pointers to multiple characters, e.g. REF ("abc") or REF ("a", "b", "c"). They can also be edited, and any DEI buffer can be saved as a text file.

By allowing pointers (and other denotable values except characters) to appear within these text files, e.g.

```
REF ("This is the referred text") d;
REF ("Here we have a link", d, "to whatever")
```

we achieve some limited form of non-linear hypertext like documents.

Other values

A textual representation of other values can also be visited by DEI. These are not very interesting, as the textual form has to be parsed to be stored as something else than text.

Versioning of other values

The expression returning a value from a versioning function need not be a function, i.e.

```
FUN /* version prelude */
-> Old &
{ Old choice;
    FUN ->.(o1 AND NOT o3 /* ... */
     NUF amb1f(choice) amb1;
    FUN /* other ambitions */
     NUF ambnf(choice) ambn
} 0;
/* versioned value */

IF 0.ambi THEN /* old changes */
  /* new E */
ELSE
  /* old E */
FI

(E1&{0 0}).E2 /* changing of environment, skipped if no changes in E2 */
E(0.choice)(E) /* calling a versioned function */
```

NUF
DEI will be equally happy to version any kind of expression following the prelude. Some possible candidates for this versioning are text files where the versioned expression is \texttt{REF ("text")}, i.e. the expression \texttt{REF ( 't', 'e', 'x', 't' )}, or environments. Some examples:

\begin{verbatim}
REF ( 't', IF 0.ambx THEN 'e' ELSE 'i' FI, 'x', 't' )
\end{verbatim}

\begin{verbatim}
{ INT a; IF 0.ambx THEN CHAR ELSE 'x' FI c; BOOL e }
\end{verbatim}

Note that as for functions, a single version is not produced before a sufficiently complete choice is applied to the versioning function.

The versioning of text files is interesting in its own right (documentation etc.), but also as a means to version test scripts and rigs.

### 4.8 Multi user control

In a Nel invocation all copies of DEI are interconnected, and cooperate to protect the users (processes) against simultaneous editing. This is similar to what is done in Emacs. The difference between DEI and Emacs is that DEI also uses the ambition and choice. Every denotable value which is visited in a buffer is recorded. If it is a versioned value and an ambition and a choice are defined, they are recorded in connection with the denotable value. For nonversioned editing the ambition and choice is symbolically set to \texttt{true}.

If the same denotable value, \texttt{D}, is visited many times (either by the same or different invocations of DEI) we have the following cases, let \texttt{B} be the new visiting buffer with choice \texttt{c_2} and ambition \texttt{a_2}:

- \texttt{c_2} overlaps with some other active choice, \texttt{c_1} for \texttt{D}. This can be visualized as the Venn diagram given in figure 4.6.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{venn_diagram}
\caption{Overlapping choices}
\end{figure}

A strong warning is given, and the user can select between different possibilities:
- Ignore the warning. This means that changes done by the buffers first committed will be cancelled by those in the following for non versioned editing. For versioned editing the visibility of the later committed changes will be restricted by those of the first, this is illustrated in figure 4.7. When a user ignores this warning, a warning is also given to the other buffers affected.

- Restrict or change \( c_2 \) or the other choice, \( c_1 \) so that the overlap disappears, i.e. select more option bindings, restrict the multi version editing.

- Cooperate on the same copy, i.e. fragments (nodes) visible with both choices will be displayed in only one buffer at a time. As said before, what to display where can be determined on the node level.

• \( c_2 \) overlaps with some active ambition, \( a_1 \) for \( D \). This means that what is seen can be changing, thus the changes that are done will not appear as they were done next time you check them out. This can be illustrated with the following example:

```
Figure 4.7: Overlapping ambition and choice
```

A warning is given, and the users have the same possibilities as above, i.e. to ignore the warning, to restrict or change the choice or ambition, or to cooperate.

• \( a_2 \) overlaps with some other active ambition, \( a_1 \) for \( D \). This means that some hidden fragment (which can only be displayed in a read only buffer) will be affected by both this and the overlapping changes, and the result depends on
the order of commitment. This is not so serious, but a warning is given, and the possibilities are the same as above, i.e. to ignore the warning, to restrict one of the ambitions, or to cooperate.

A note on cooperation

When two or more buffers are cooperating on the same value with different ambitions, the cooperation is restricted so that any resulting choice will be syntactically correct.

To guarantee this we make the following two restrictions:

- Any fragment can only be changed (moved or deleted) in one buffer. When it is changed it is locked to this buffer. We can lock fragments before they are changed. Locked (changed) fragments can be given to another cooperating buffer, which means that the ambition of that change will be that of the receiving buffer.

- One buffer can only be committed when all the cooperating buffers are syntactically correct. Only the changes done in the committed buffer will be committed. These changes are automatically integrated in the remaining cooperating buffers.

The problems we have avoided are the following:

- If arbitrary tokens are changed with different ambitions, e.g. \(3+4\) is changed to REF \(b\), by changing \(3\) to REF in one buffer and \(+4\) to \(b\) in another, the result outside to overlap of the ambitions will be syntactically incorrect, i.e. REF \(+4\) and \(3\ \ b\).

- A complex representation of visibilities if one fragment is changed from many buffers is the result of changing a fragment in many buffers. Consider again the example from figure 4.7, but where either \(B_1\) changes or \(B_2\) changes \(E\) to \(E'\).

  - \(B_1\) changes \(E\) to \(E'\). \(E'\) will become the new shared node, i.e. \(E\) in \(B_2\) is changed to:

    \[
    \text{IF } a_1 \text{ THEN } E' \text{ ELSE } E \text{ FI} \\
    \text{reduced to } /*(*)/ \ E' /*(*)/*/ \text{ or } E'
    \]

    What is displayed in \(B_2\) is also changed from

    \[
    /* \text{buffer1:nummer1 }*/ \ /* \text{before }*/ \\
    /* \text{buffer1:nummer2 }*/ \ /* \text{after }*/
    \]

    where \text{number1} referred to the node number of \(E\), while \text{number2} refers to the node number of \(E'\). Later changes to \(E'\) by \(B_2\) will only get visibility \(a_1 \land a_2 \land v\).
- $B_2$ changes $E$ to $E'$. $E$ can cease to be shared, and the original $E$, which was displayed as

    /* buffer2: number1 */

    can appear as

    IF a2 THEN /* buffer2: number2 */ ELSE E FI

    reduced to /*(*) E /**/)*/ or E

    in $B_1$ indicating that it is not longer shared. /* buffer2: number2 */ refers to $E'$. Any changes done to $E$ in $B_1$ will get the visibility $a_1 \land \neg a_2 \land v$, and not affect $E'$ which has visibility $a_1 \land v$.

Thus we would have to record which buffers (ambitions) had already changed a node, which complicates the representation. Now we know that all expressions which are locked to a buffer are only changed with its ambition.

A note on changing choice and ambition

As we have seen the ambition determines the scope of the changes, whereas the choice determines what is seen and thus can be changed.

Non cooperative buffers can change their ambition as long as it does not affect any other choice or ambition for the same value. Changing the choice (within the ambition) before committing is done by integrating the changes in the copy, and making a new edit copy by restricting the result with the new choice. Note that the integration must produce something syntactically correct, i.e. a new choice can not be selected before the current one is consistent.

For cooperative buffers things are more complicated. Here the ambition can be expanded as long as it does not affect any choice or ambition in buffers outside the cooperating group. If two buffers had the same ambition, they also shared the locks, which have to be divided if one of the buffers changes its ambition. Restricting an ambition can make cooperative buffers non cooperative, thus the locks on the shared nodes will be removed, and the original value of the node will be fetched from the copy. Changing the choice within a cooperative buffer works as for non cooperating buffers, only the buffer changing the choice need to be syntactical correctness.

When one of the buffers are merged with the original file, the others are notified. They can now choose to merge the new changes into their own copy, or they can continue as before.

A note on merging

When a value is visited in two buffers we have to record changes from them both. This can be illustrated with the following figure:
Each copy \((e_i)\) has the numbering of the original, i.e. \(n_0\). When \(e_1\) is merged with the original the original is renumbered to identify the common nodes, and we make a new tree (dag) \(c_1\) containing the ambition \(a_1\).

When the merged value \(c_1\) is written over the original (the changes are committed), the ambition \(a_1\) is propagated to the other buffers containing a copy, and a notifier is given to those with overlapping choice or ambition. They can choose if they want to incorporate the changes, or continue without them. The incorporation of changes is only interesting if there is some overlap between the ambitions or choices of the two buffers, otherwise the changes will not affect each other. If it is decided that the changes should be incorporated the following procedure is executed:

1. Copy the new original \((c_1)\) and enumerate it giving a new numbering \(n_1\). 
   Set \(a_1\) to false and recreate \(c_0\) retaining the new numbering \(n_1\). Make a second numbering \(n_0\) of \(c_0\) which is the same as the numbers in \(e_2\), and make the mapping \(m_{0\to1} : n_0 \to n_1\).

2. Integrate \(e_2\) and \(c_1\), using the mapping \(m_{0\to1}\) to identify nodes. This gives a tree \(c_2\), with the numbering \(n_1\) on the nodes.

3. Make a new editing copy out of \(c_2\) by using the choice and deleting non visible nodes. Retain the \(n_1\) numbers on the visible nodes. Discard the tree \(c_0\), the numbering \(n_0\) and the mapping \(m_{0\to1}\), which were only temporary.

4. Record that \(a_1\) is integrated for this buffer, and that the numbering used is \(n_1\).

If the changes from 2 shall be directly committed into the changed original, step 1 and 2 above are done, followed by:

3. Writing \(c_2\) over the original, and propagating \(a_2\).

A third buffer having a copy of the original tree \(c_0\) will now have received both ambition \(a_1\) and \(a_2\) which must both be set to false to find the numbering \(n_0\) from \(c_2\).
A note on merging The third buffer can now integrate either the changes from the first or the second buffer with its own editings $e_3$, i.e. to first integrate $a_2$ and then $a_1$, the following is done:

1. Fetch $c_2$, and enumerate it making $n_2$.
2. Set $a_1$ and $a_2$ to false recreating $c_0$.
3. Make $m_{0-2}$ and integrate $c_2$ with $e_3$, making $c_3$.
4. Set $a_1$ to false, and use the choice to make a new editing copy out of $c_3$ retaining the numbers from $n_2$ on the nodes.
5. Record that $a_2$ is integrate for this buffer, and the number used is $n_2$.
6. Integrate $a_1$ to make a new $c_3$ by integrating $e_3$ directly in $c_2$. $c_3$ can now be committed, or editing can continue by using the choice to convert it into an editing copy.

A note on the non reflexivity of mergings When we are visiting a value in two buffers with non overlapping ambitions, the order of merging (commitment) is of importance for the visibility. Consider the following example:

$$a_1 : t \Rightarrow t_1 \quad \text{IF} \quad a_1 \quad \text{THEN} \quad t_1 \quad \text{ELSE} \quad \text{IF} \quad a_2 \quad \text{THEN} \quad t_2 \quad \text{ELSE} \quad t \quad \text{FI} \quad \text{FI}$$
$$a_2 : t \Rightarrow t_2 \quad \text{IF} \quad a_2 \quad \text{THEN} \quad t_2 \quad \text{ELSE} \quad \text{IF} \quad a_1 \quad \text{THEN} \quad t_1 \quad \text{ELSE} \quad t \quad \text{FI} \quad \text{FI}$$
$$c_1 \rightarrow a_2$$

Here $a : t \Rightarrow t'$ means that $t$ is changed to $t'$ under ambition $a$, and $c \rightarrow a$ means that $c$ is within the ambition $a$. If $a_2$ commits before $a_1$, and we immediately visit the value with the same choice $c_1$ we will have the traumatic exprience that $t_1$ is replaced by $t_2$! A warning of this can be given by reevaluating the choice before $a_1$ is committed, and if it is different from the edited copy, the difference can be recorded as editing commands. $a_1$ is not committed, but the merged $a_1$ and $a_2$ are made the new editing copy.

Low level manipulation of the version structure In principle nothing can prevent the user from visiting the raw multiversion function, i.e. the prelude and all the versioning code. This is not recommendable, and will duplicate the dag structure of the multi version file to a syntax tree. If the function is written back the sharing information is lost. We can still do some dirty tricks. In multiversion editing, the ambitions are shown, and can be edited. This is useful for giving the changes in one branch wider visibility at the expense of the changes in the other other branch. E.g. changing IF $a_1$ THEN ... to IF $a_1$ AND $a_2$ THEN ... with ambition $a_3$ will have the desired effect and be recorded as IF IF $a_3$ THEN $a_1$ AND $a_2$ ELSE $a_1$ FI THEN ... . DEI will not use the new ambition to determine if only one branch is visible before next time the choice is evaluated. With the ambition TRUE and this trick we can achieve anything. E.g. Smith wants to delete all changes done by Jones with ambition $a_5$, which he does by replacing $a_5$ with FALSE under ambition TRUE. If Jones wants them back he does the same thing, i.e. replacing
IF TRUE THEN FALSE ELSE a5 FI with a5, under the same ambition. Instead of continuing this rather stupid activity Smith and Jones should agree on introducing a new option so that they can choose between including the the changes a5 or not. Note that no simplification is done with an empty choice environment.

The use of options

Options are used for two main purposes:

- To record parallel variants, which is most commonly done with conditional compilation, variant records or subclasses in traditional systems.

- To record sequential revisions, which is most commonly done with systems like SCCS in traditional systems.

A higher level interface to change oriented versioning is described in [GULLA90].
Chapter 5

Semantics

5.1 Introduction

This chapter contains the definition of the Nel language. It is given in a denotational style with notation taken from [STOY77]. The semantic definition is meant to give a formal definition of the meaning of every language construct, and be a guideline for an implementation.

The semantics of the sequential part of Nel is given in section 5.2. It is given as standard continuation semantics, where the static bindings of the \( \lambda \)-calculus is used to keep track of the stacks of denotable values, environments and continuations. Section 5.3 gives the semantics for the parallel constructs in an interleaving style, i.e. the meaning of a parallel program is all possible interleavings. Section 5.4 indicates how the semantic equations have to be changed to explicitly represent the denotable value, environment and continuations stacks, instead of implicitly representing them in the \( \lambda \)-expressions. This representation is necessary to treat unification of continuations correctly, but is not worked out in the semantic equations. It is also necessary for the abstract interpretation which is indicated in chapter 6.

The semantic definition given here is rather operational, i.e. being very near to an interpreter for the language. All the complex operations like subset test and unification are given in detail, and do of course only represent one alternative on how to implement such algorithms. Some of the algorithms are perhaps also to complex to be suitable for a presentation coded in \( \lambda \)-calculus.

Splitting the semantic definition into a sequential and a parallel part turned out to be rather fortunate, as there is little connection between these parts when the atomic actions are defined. This simplified the description of the sequential part substantially.
5.2 Sequential semantics

5.2.1 Notation

The notation is used in the definition is summarized below. It is fairly standard, and similar to what is used in e.g. [STOY77] and [REES86]. Note that some of the symbols are overloaded, but the use should be apparent from the context.

\textbf{Power}(A) \quad \text{the powerdomain of } A
\begin{align*}
A \cup B & \quad \text{union of domains} \\
\begin{array}{ll}
a + b & \quad \text{addition on integers} \\
strut & \quad \begin{align*}
a + b & \quad \text{concatenation of environments, } b \triangleleft a \text{ or } \lambda l.(b l = \bot) \rightarrow a l, b l \\
a - b & \quad \text{subtraction on integers} \\
a \triangleright b & \quad \text{difference of environments, } a \triangleright \# b \\
A \rightarrow B & \quad \text{domain of continous functions from } A \text{ to } B \\
a \rightarrow b, c & \quad \text{McCarthy conditional} \\
A \times B & \quad \text{Cartesian product of domains} \\
\begin{array}{ll}
a \times b & \quad \text{multiplication on integers} \\
A^+ & \quad \text{domain of non empty sequences of elements from } A \\
A^* & \quad \text{domain of possibly empty sequences of elements from } A \\
\langle \ldots \rangle & \quad \text{sequence formation} \\
(a, b) & \quad \text{product formation, i.e. } (a, b) \in A \times B \\
s \downarrow k & \quad k\text{th member of the sequence } s, 1\text{-based} \\
\# s & \quad \text{length of sequence} \\
s \triangleleft t & \quad \text{concatenation of sequences} \\
s \triangleright k & \quad \text{drop the first } k \text{ members of } s \\
m l & \quad \text{function application.} \\
m[l/v] & \quad \text{substitution, i.e. } \lambda a.(a = l) \rightarrow v, m l \\
v \in A & \quad \text{true if } v \text{ is an element of the domain } A \\
A \sqsubseteq B & \quad \text{true if } A \text{ is a subdomain of } B
\end{array}
\end{align*}
\end{align*}

In the semantic description we do not distinguish between the singleton domain, \{v\}, and its value, v. To be correct we would have to use injection and projection functions all around.

Functions can be represented as lists \((\text{I} \text{de}, D \text{Val}^+)\) or as continuous functions \(\text{I} \text{de} \rightarrow D \text{Val}^+\), and these representations will be interchanged.

The symbol \(\rightarrow\) for constructing domains of continuous functions is right associative, i.e. \(A \rightarrow B \rightarrow C\) is \(A \rightarrow (B \rightarrow C)\). The ordinary \(\lambda\)-calculus is used as the meta language. Function application is left associative, i.e. \(abc = ((ab)c)\).

The semantic equations are written on the following form:

\[E[E_1 \text{ op } E_2 | c] = \lambda \text{state}.E[E_1 | c_1 \text{ state}_1]
\]
\[c_1 = \lambda \text{state}_1'. \ldots
\]
\[
\begin{aligned}
\ldots \\
c_i = \lambda \text{state}_i'. \ldots c \text{ state}_{i+1}
\end{aligned}
\]

state is here just an abstraction for the intermediate result of the computation.
The sequencial steps are factored out in separate continuation, i.e. \( c_i = \ldots \). To get the correct form they should be textually substituted for their occurrences.

The semantic equations are recursive, e.g. \( x = fx \). The fix point combinator

\[
Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\]

can be used to define these fixpoints, i.e. \( Yf = x \).

Each semantic definition is proceeded by a short informal description, highlighting interesting points.

**Why is not Nel used to define Nel?** This might seem like a relevant question, and it could easily be done, actually the continuation based semantics defined here maps easily into Nel. The only reason for not doing so, is that the representation here seems more abstract, and we can distinguish more clearly between the levels.

### 5.2.2 Sequential semantic domains

This section describe the semantic domains for the sequential part of Nel. After each domain a short description of their use is given.

**Values**

\[
\begin{align*}
v \in Val & = \text{Power}(\text{Int} \cup \text{Bool} \cup \text{Char} \cup \{\text{nil}\} \cup \text{DVals} \cup \text{Env} \cup \text{Fun} \cup \text{Contv}) \\
\text{Bool} & = \{\text{true, false}\}
\end{align*}
\]

\( Val \) is the domain of values, which is the powerset of all primitive constructable values. \( \text{Int} \) and \( \text{Char} \) are defined similarly to \( \text{Bool} \) and are just abbreviations for listing up all the elements.

**Denotable values**

\[
\begin{align*}
d \in DVal & = (\text{Loc} \times \text{Access}) \\
l \in Loc & = \text{Int} \\
a \in Access & = \{\{\text{limit, const}\}\} \\
e \in DVals & = DVal^+
\end{align*}
\]

\( DVal \) is the domain of denotable values, i.e. those returned from expressions. They contain a location, and access information. \( DVals \) are multiple denotable values.

**Environments**

\[
r \in Env = \text{Id} \rightarrow (DVal^+ \cup \perp)
\]

\( Env \) is the domain of environments, which is the domain for both the current environment, and environment values.
Store

\[
m \in \text{Store} = \text{Loc} \to (\text{MaxVal} \times \text{ActVal})
\]
\[
t \in \text{MaxVal} = \text{Val}
\]
\[
v \in \text{ActVal} = \text{Val}
\]

*Store* is the mapping from locations to actual and maximal value. As we are most often interested in either the actual (value) or the maximal (type) value of a location, we define two access functions \(m_v\) and \(m_t\) to get at these values, i.e. let \(m_t = \lambda(m,n)(\lambda(t,v).t)(ml)\). Getting the actual value of a location is then \(m_v sl\).

Functions and continuations

\[
c \in \text{Cont} = \text{DVals} \to \text{Env} \to \text{State} \to \text{State}
\]
\[
\text{Contv} = \text{Cont} \times \text{Env} \times \text{Id}
\]
\[
\text{Fun} = (\text{Cont} \to \text{Cont}) \times \text{Env} \times \text{Id}
\]
\[
\text{Id} = \text{Int}
\]

*Cont* is the domain for the rest of the program (execution). Continuation values *Contv* and functions *Fun* have in addition to the code, the static environment and unique identifier explicitly represented.

State

\[
s \in \text{State} = (\text{Store} \times \text{Next}) \cup \bot
\]
\[
n \in \text{Next} = \text{Next}_t \times \text{Next}_f
\]
\[
n_t \in \text{Next}_t = \text{Int}
\]
\[
n_f \in \text{Next}_f = \text{Int}
\]

*State* contains the store, the next free location and the next continuation / function identifier. \(\bot\) is the error state.

The letters listed in the left column are those used for variables which should denote values from the corresponding domains in the semantic equations.

### 5.2.3 The semantic function \(E\)

The meaning function \(E\) maps each syntactically correct Nel expression into a function on the semantic domains. The definition of \(E\) is quite standard:

\[
E : \text{Exp} \to \text{Cont} \to \text{DVals} \to \text{Env} \to \text{State} \to \text{State}
\]

### 5.2.4 Recursive values

The complexity in the Nel value structure stems from \(\text{Env}\) (*DVal* is just a special case of \(\text{Env}\) as explained in section 2.3.3). If we abstract all the non recursive
values into \textit{PVal} (primitive values), disregard access information, maximal and actual values, and represent pointers as environments, we can represent the store as:

\[ \text{Store} = \text{Loc} \rightarrow \text{Power}((\text{Id}, \text{Loc}^*)^* \cup \text{PVal}) \]

The ordering of \textit{PVal} is trivial when we have made all functions and continuations distinct with unique identifiers \textit{(Id)}. For environments one environment \(x\) is a subset of another environment \(y\) if \(x\) contains all the structure of \(y\) and possibly more, i.e.

- The declarations of \(y\) occurs in \(x\), and \(x\) might have more declarations.
- For each common declaration the multiplicity of the locations is greater or equal for \(x\) than for \(y\).
- The value of each common location from \(x\) is a subset of that from \(y\).
- \(x\) contains all the aliases of \(y\), and possibly more.

The algorithm of unification can now be described as: Unification of \(u\) and \(v\) is the greatest structure which is a subset of both.

In the semantic definition algorithms for constructing these structures are given. These algorithms assume that we can retrieve each non union constructable component of a value as a list, i.e. \(\text{list} : \text{Val} \rightarrow \text{Val}^*\). Since every element of \textit{Val} is constructed from primitive elements, can we represent elements of \textit{Val} as such a list of primitive values.

There is one problem with values filling up a set value, e.g. \texttt{TRUE UNION FALSE} should be the same as \texttt{BOOL}, thus after each extension of a union, a check should be made to see if this happens. This check has also to be done on pointers and environments, e.g.

\[
\begin{align*}
\text{REF TRUE UNION REF FALSE} &= \text{REF BOOL}; & /* \text{TRUE */} \\
\{\text{TRUE a}\} \text{ UNION } \{\text{FALSE a}\} &= \{\text{BOOL a}\}; & /* \text{TRUE */}
\end{align*}
\]

This check is necessary for \textit{Bool}, \textit{Int} and \textit{Char}, and only of practical importance for \textit{Bool}.

The algorithm also becomes simpler if we assume that \textit{list} gives out the list in some predefined order. One plausible order might be:

1. All \textit{Int}, \textit{Bool} and \textit{Char}, e.g. \((4, 5, \textit{Bool}, c)\).

2. All environments ordered on lexicographical order of names, and in the case of equal names ordered on the values of the first unequal component, e.g.
3. All pointers ordered on the multiplicity of the value, and if equal the referenced value (maximal), e.g. \texttt{REF INT}, \texttt{REF TRUE}, \texttt{REF REF TRUE}, \texttt{REF (INT, INT)}.

4. All functions and continuations in order of creation, \texttt{Id} identifies all functions and continuations.

In the algorithms we will also assume that we have operations to check the equality \(=\) and subset \(\subseteq\) relations, and construct the intersection \(\sim\) of subsets of \(PVal\), i.e. all values except pointers and environments.

### 5.2.5 Auxiliary functions

The auxiliary functions are like a subroutine library which is used in the semantic equations.

They are defined on the following form:

\[
\begin{align*}
\text{name} & : A \rightarrow B \\
\text{name} & = \lambda a. \text{body}
\end{align*}
\]

i.e. first giving the type of the function and then the definition. Each auxiliary function is followed by a short description. The description only comments values changed by the function.

Note that most auxiliary functions are “continuation transformers”, i.e. they take the rest of the program as a parameter, and passes the result to that. This makes it simpler to treat error conditions.

Some auxiliary functions are also defined in connection with each definition.

### Apply

\textit{Apply} take one or two lists, and a function, applies that function to each element, and returns the resulting list. There are many other forms of apply scattered around in the semantic equations, each of them applying functions taking different arguments and returning different values.

\[
\begin{align*}
\text{apply}_1 & : \text{Cont} \rightarrow \text{Fun} \rightarrow X^* \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{State} \\
\text{apply}_1 & = \lambda cfrs.(\#l = 0) \rightarrow c()rs, fc_1(l \downarrow 1)rs \\
\text{c}_1 & = \lambda l_1rs.\text{apply}_1c_2f(l_1\uparrow 1)rs \\
\text{c}_2 & = \lambda l_2rs.c(l_1\uparrow l_2)rs
\end{align*}
\]
apply_2 : Cont → Fun → X* → X* → Env → State → State
apply_2 = λe₁l^1l^2rs.((#l^1 = 0) → c(())rs, fc_1(l^1↓1,l^2↓1)rs
     c_1 = λl_1rs.apply_2 c_2 f(l^1↑1)(l^2↑1)rs
     c_2 = λl_2rs.c(l_1l_2rs)

Allocating a new location

New takes a value v, and a store, m, and finds an unused location which gets
maximal and actual value v. The DVal consisting of the location together with an
empty set of access codes, and the new store are passed on.

new : Cont → Val → Env → State → State
new = λcur(m,n_1,n_f).c((n_1,{}))r(m_1,n_1+1,n_f)
m_1 = λl.(l = n_1) → (v,v), ml

Some predefined denotable values

Error messages are given as text, i.e. reference to a multiple value of characters.
These are built up once and for all, and are available in the error environment,
or as e_text in the semantic descriptions. We define e_overflow as an example:

CONST REF CONST ("overflow") overflow;

Any syntactic expression occurring in the program, e.g. 4+3 is also available as a
denotable text value with line number giving its relative offset from the enclosing
function: e_3+4

CONST REF CONST ("3 + 4 /* line 10 */")

5.2.6 Exceptions

Handling of exceptions is divided into two phases:

- Building up the return value. Here it is one function for each exception.
  These functions are totally similar except that the text for the exception
  changes, one example is accessing a limited denotable value:

limited : Cont → DVals → Env → State → State
limited = λcur(m,n_1,n_f).apply_1 new c_1
     (((λe_drs.raise c limited (e_d↓1)e_prs),r,n_f),c,r,n_f+1,e,r)
     r(m,n_1,n_f+2)
     c_1 = λers.c_2(eₘₑₘₑₘₑₘₑₘₑₘₑₘₑₘₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑ’e_p
Note here that we have used e_p before we have constructed it, thus the
definition is recursive. The definition of these functions is not repeated for
each exception.
• Searching for the handler which is common for all exceptions:

\[
\begin{align*}
\text{raise} & : \text{Cont} \to \text{Ide} \to \text{DVals} \to \text{Env} \to \text{State} \to \text{State} \\
\text{raise} & = \lambda c e p r s . ((r \text{TRAP}) = \perp) \to \perp, \text{raise}_1 e c l (r \text{TRAP} \downarrow 1) e p r s \\
\text{raise}_1 & = \lambda c l (l, a) e p r s . (m_v sl \in \text{Env}) \to c_2 (m_v sl) e p r s, \perp \\
c_2 & = \lambda r_1 r s . (r_1 l = \perp) \to \\
& \quad ((r_1 \text{catchall}) = \perp) \to \perp, c_3 ((r_1 \text{catchall}) \downarrow 1) r s, \perp \\
c_3 & = \lambda (l, a) r s . (m_v sl \in \text{Contv}) \to c_4 (m_v sl) e p r s, \perp \\
c_4 & = \lambda (c, r_c, n_f) e p r . c e p r s \\
\end{align*}
\]

5.2.7 Type checking functions

There are two kinds of type checking functions, those which do check if the denotable value is limited, and those which do not. Those which do not are \textit{isFun}, \textit{isCont} \textit{isDVal} and \textit{isEnv}. Concatenation on environments are only allowed if the environments are not limited, thus for environments we also have a check including the check for limited denotable value, i.e. \textit{isEnv}.

The following auxiliary function, \textit{isInt}, is used to check if a denotable value is not limited and an integer, and if not it raises the appropriate exceptions.

\[
\begin{align*}
\text{isInt} & : (\text{Val} \to \text{Env} \to \text{State} \to \text{State}) \to \text{DVals} \to \text{Env} \to \text{State} \to \text{State} \\
\text{isInt} & = \lambda c l (l, a) r s . (\text{limit} \in a) \to \text{limited} c_n ((l, a)) r s, c_1 (l, a) r s \\
c_1 & = \lambda (l, a) r s . (m_v sl \in \text{Int}) \to c((l, a)) r s, \text{notINT} c_n (m_v sl) r s \\
c_n & = \lambda e r s . (\text{isInt} c)(e \downarrow 1) r s \\
\end{align*}
\]

The continuation \(c\) takes a value instead of a denotable value as argument. The continuation \(c_n\) to both exceptions uses only one denotable value, and starts at the limited check.

\textit{isFun} is defined similarly:

\[
\begin{align*}
\text{isFun} & = \lambda c l (l, a) r s . (m_v sl \in \text{Fun}) \to c(m_v sl) r s, \text{notFUN} c_n ((l, a)) r s \\
c_n & = \lambda e r s . (\text{isFun} c)(e \downarrow 1) r s \\
\end{align*}
\]

5.2.8 Simple values and their operations

Primitive values

Primitive values are evaluated by passing the corresponding set to \textit{new}.

\[
\begin{align*}
\text{E}[\text{VOID}] c & = \lambda e r s . \text{new } c \text{VOID } r s \\
\text{E}[\text{TRUE}] c & = \lambda e r s . \text{new } c \text{TRUE } r s \\
\text{E}[\text{FALSE}] c & = \lambda e r s . \text{new } c \text{FALSE } r s \\
\text{E}[\text{N}] c & = \lambda e r s . \text{new } c \text{N } r s \\
\text{E}[\text{C}] c & = \lambda e r s . \text{new } c \text{C } r s \\
\text{E}[\text{INT}] c & = \lambda e r s . \text{new } c \text{INT } r s \\
\text{E}[\text{CHAR}] c & = \lambda e r s . \text{new } c \text{CHAR } r s \\
\text{E}[\text{DIOV}] c & = \lambda e r s . \text{new } c \text{DIOV } r s \\
\end{align*}
\]

88
True is the set \{true\}, \textit{4} is the set \{ 4 \}, \textit{Diov} is the set of all values, \textit{Void} is the empty set, and so on.

**Operations on primitive values**

The operations on primitive values are quite similar. First \( E_1 \) is evaluated, then \( E_2 \), then a check is made that the results have the same number of multiplicity before the operation is applied to each pair.

\[
\begin{align*}
E[E_1 + E_2]c &= \lambda e rs. E[E_1]c_1 rs \\
& \quad \lambda e_1 rs. E[E_2]c_2 rs \\
& \quad \lambda e_2 rs. (#e_1 = #e_2) \rightarrow \text{apply}_2 c \ add e_1 e_2 rs, \ new c_3 e_1 rs \\
& \quad \lambda e_1 rs. \ new c_4 e_2 rs \\
& \quad \lambda e_2 rs. \text{diffrange} c(e_1, e_2) rs
\end{align*}
\]

Note here that the continuation passed to the \textit{diffrange} exception is the rest of the program.

The equations for \( E - E, E \times E, E \div E, E \mod E, E \text{ AND } E, E \text{ OR } E \) and \textit{NOT E} are very similar, and are not given. Numeric comparison, i.e. \( E \leq E \) and \( E < E \) will also fail for multiple values of different length.

**Auxiliary functions on Int**

The syntactic operators \(+, -, *, /, \mod, \leq, <\) on \textit{INT} have their semantic counterparts on singleton subsets of \textit{Int}. Operations on integers can also raise the exceptions \textit{overflow}, \textit{divbyzero}, \textit{limited} and \textit{notINT}.

The following functions are defined:

\[
\begin{align*}
\text{add} & : \text{Cont} \rightarrow (DVal \times DVal) \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{State} \\
\text{add} &= \lambda c(d_1, d_2)rs. \text{isInt} c_1 d_1 rs \\
c_1 &= \lambda v_1 rs. \text{isInt} c_2 d_2 rs \\
c_2 &= \lambda v_2 rs. (\text{minint} \leq (v_1 + v_2) \leq \text{maxint}) \rightarrow \\
& \quad \\new c(v_1 + v_2)rs, \\
& \quad \text{overflow} c_n(d_1, d_2, d_+) rs \\
c_n &= \lambda rs. c(e \downarrow 1) rs
\end{align*}
\]

The continuation, \( c_n \) after overflow is the rest of the program, thus a new value can be given for the overflow result.

\textit{sub} and \textit{mult} are similar to \textit{add} only that \(+\) is replaced by \(-\) and \(*\) respectively. \textit{div} and \textit{mod} are a bit different as they check for \textit{0}, i.e.:
\[ \text{div} : \text{Cont} \rightarrow (DVal \times DVal) \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{State} \]
\[
\text{div} = \lambda c (d_1, d_2) rs. \text{isInt} \ c \ dt_1 \ rs
\]
\[
c_1 = \lambda v_1 rs. \text{isInt} \ c \ dt_2 \ rs
\]
\[
c_2 = \lambda v_2 rs. (v_2 = 0) \rightarrow
\]
\[
divbyzero c_n (d_1, d_2, d_{\text{divbyzero}}) rs,
\]
\[
\text{new} \ e (v_1 / v_2) rs
\]
\[
c_n = \lambda e rs. e (\bot) rs
\]

\text{mod} \ is \ similar \ to \ /.

\text{leq} \ and \ \text{le} \ to \ check \ if \ an \ integer \ is \ less \ than \ another \ are \ defined \ as \ follows:

\[ \text{leq} : \text{Cont} \rightarrow (DVal \times DVal) \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{State} \]
\[
\text{leq} = \lambda c (d_1, d_2) rs. \text{isInt} \ c \ dt_1 \ rs
\]
\[
c_1 = \lambda v_1 rs. \text{isInt} \ c \ dt_2 \ rs
\]
\[
c_2 = \lambda v_2 rs. ((v_1 \leq v_2)) \rightarrow \text{new} \ e \ True \ rs, \text{new} \ e \ False \ rs
\]

\text{le} \ is \ similar.

**Auxilliary functions on \textit{Bool}**

The syntactic operators \textit{AND}, \textit{OR} and \textit{NOT} on \textit{BOOL} have their semantic counterpart on \textit{Bool}. Operations on booleans may raise the exceptions \textit{limited} and \textit{notBOOL}.

The auxilliary function \textit{isBool} to check if a denotable value is a boolean and not limited is similar to \textit{isInt} defined for integers.

The following functions are defined:

\[ \text{and} : \text{Cont} \rightarrow (DVal \times DVal) \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{State} \]
\[
\text{and} = \lambda c (d_1, d_2) rs. \text{isBool} \ c \ dt_1 \ rs
\]
\[
c_1 = \lambda v_1 rs. \text{isBool} \ c \ dt_2 \ rs
\]
\[
c_2 = \lambda v_2 rs. ((v_1 = \text{true}) \land (v_2 = \text{true})) \rightarrow \text{new} \ e \ True \ rs, \text{new} \ e \ False \ rs,
\]

\text{or} \ and \ \text{not} \ are \ similar.

**5.2.9 Declarations**

Declarations and use of identifiers use the current environment \(r\). The equations are:

\[
E[E: \text{Idlist}]c = \lambda e rs. E[E: \text{Idlist}]c e rs
\]
\[
c_1 = \lambda e_1 rs. (\#e_1 < \#\text{Idlist}) \rightarrow \text{new} c e_1 rs, \text{apply}_2 c \text{decl} \ e \text{Idlist} rs
\]
\[
c_2 = \lambda e_1 rs. \text{isrange} e_1 \text{Idlist} e rs
\]
\[
\text{decl} = \lambda c \text{cdr} e rs \text{dr} \|/d\|rs
\]

\[
E[\text{UNDEF} \text{Idlist}]c = \lambda e rs. \text{apply}_1 c e \bot \text{Idlist} rs
\]
\[
\text{decl} = \lambda c \text{cls} \text{rs} \text{dr} \|/\bot\| rs
\]
\[
c_1 = \lambda e rs. c e rs
\]
\[ E[ISDEF \text{ Idlist}]c = \lambda e.\text{apply}_1c \text{ isdef Idlist}rs \]
\[ \text{isdef} = \lambda c_1 rs.(rl = \perp) \rightarrow \text{new } c \text{ False}rs, \text{new } c \text{ True}rs \]
\[ E[\text{BEGIN } E \text{ END}]c = \lambda e.\text{E}[E]c_1 rs \]
\[ c_1 = \lambda e_1 rs \]
\[ E[I]c = \lambda rs.(r = \perp) \rightarrow \text{undefined } c_1 rs, c(r)rs \]

The initial environment is defined as \((\lambda l. \perp)\)

### 5.2.10 Constructed values

**Pointers**

Pointers are constructed by giving the denotable value to `new`, and destructed by using the store to retrieve the denotable value. Following a pointer can raise the exceptions `notREF`, `limited` and `nilpointer`.

\[ E[\text{REF } E]c = \lambda e.\text{E}[E]c_1 rs \]
\[ c_1 = \lambda e_1 rs \]
\[ E[\text{NIL}]c = \lambda e.\text{new } c \text{ nil}rs \]
\[ E[\text{DEREF } E]c = \lambda e.\text{E}[E]c_1 rs \]
\[ c_1 = \lambda e_1 rs \text{ deref}ers \]
\[ \text{deref} = \lambda c_2 ds.e \text{ isDVal } c_2 rs \]
\[ c_2 = \lambda rs.(v = \text{nil}) \rightarrow \text{nilpointer } c(d)rs, c(rs) \]

**Multiple values**

Multiple values are only operating on the list of denotable values, concatenating, giving the length and selecting. Note that a multiple denotable value can be given as the selector, e.g.:

\[ 3, 4, 2, 5[2, 3, 2] \quad /\* 4, 2, 4 /\* \]

is allowed.
\[ E[E_1, E_2]c = \lambda e_{\text{rs}}.E[E_1]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.E[E_2]c_2ers \]
\[ c_2 = \lambda e_{\text{rs}}.c(e_1 \downarrow e_2)rs \]

\[ E[\text{LEN } E]c = \lambda e_{\text{rs}}.E[E]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.\text{new } c(\#e)rs \]

\[ E[E_1[E_2]]c = \lambda e_{\text{rs}}.E[E_1]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.E[E_2]c_2ers \]
\[ c_2 = \lambda e_{\text{rs}}.\text{apply}_1c \text{ select } e_2rs \]
\[ \text{select} = \lambda e_{\text{rs}}.\text{isInt } c_3drs \]
\[ c_3 = \lambda e_{\text{rs}}.(1 \leq v \leq \#e_1) \rightarrow c((e_1 \downarrow v)rs, \text{new } c_4e_1rs \]
\[ c_4 = \lambda e_{\text{rs}}.\text{indexrange } c(d, e_1^r)rs \]

**Environments**

The operations on environments are construction, opening and concatenation.

\[ E[\{E\}]c = \lambda e_{\text{rs}}.E[E]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.\text{newc}(r_1 - r)rs \]

\[ E[\text{THISENV}]c = \lambda e_{\text{rs}}.\text{newerrs} \]

\[ E[E_1, E_2]c = \lambda e_{\text{rs}}.E[E_1]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.\text{apply}_1c_2\text{newenv } e_1r_1s \]
\[ c_2 = \lambda e_{\text{rs}}.E[E_2]c_3ers \]
\[ c_3 = \lambda e_{\text{rs}}.c_4ers \]
\[ \text{newenv} = \lambda e_{\text{rs}}.\text{isEnv } c_4drs \]
\[ c_4 = \lambda e_{\text{rs}}.c_5 \]

\[ E[E_1 & E_2]c = \lambda e_{\text{rs}}.E[E_1]c_1ers \]
\[ c_1 = \lambda e_{\text{rs}}.E[E_2]c_3ers \]
\[ c_2 = \lambda e_{\text{rs}}.c(\#e_1 = \#e_2) \rightarrow \text{apply}_2c_3\text{addenv } e_1e_2rs, \text{new } c_4e_1rs \]
\[ c_3 = \lambda e_{\text{rs}}.\text{newc } c_4e_2rs \]
\[ c_4 = \lambda e_{\text{rs}}.\text{differ } c(e_1, e_2)rs \]
\[ \text{addenv} = \lambda e_{\text{rs}}.\text{isEnv } c_6d_1rs \]
\[ c_5 = \lambda e_{\text{rs}}.\text{isEnv } c_6d_2rs \]
\[ c_6 = \lambda e_{\text{rs}}.\text{newc } (v_1 + v_2)rs \]

We will here suppose that when a new environment value is made, the visible declarations are sorted in the lexicographical order.

### 5.2.11 Access control

Access control restricts the access to denotable values by making a new denotable value where the access information is extended. There are also operations to check the accessability of denotable values.
\[ E[\text{CONST } E|c] = \lambda\text{ers}. E[E|c_1] \text{ers} \]
\[ c_1 = \lambda\text{ers}.\text{apply}_1c\ makeconsters \]
\[ \text{makeconst} = \lambda c(l, a)rs.c(l, a \cup \{\text{const}\})rs \]

\[ E[\text{LIMIT } E|c] = \lambda\text{ers}. E[E|c_1] \text{ers} \]
\[ c_1 = \lambda\text{ers}.\text{apply}_1c\ makelimiters \]
\[ \text{makelimit} = \lambda c(l, a)rs.c(l, a \cup \{\text{limit}\})rs \]

\[ E[\text{ISCONST } E|c] = \lambda\text{ers}. E[E|c_1] \text{ers} \]
\[ c_1 = \lambda\text{ers}.\text{apply}_1c\ checkconsters \]
\[ \text{checkconst} = \lambda c(l, a)rs.(const \in a) \rightarrow \text{new } c \text{ True } rs, \text{new } c \text{ False } rs \]

\[ E[\text{ISLIMIT } E|c] = \lambda\text{ers}. E[E|c_1] \text{ers} \]
\[ c_1 = \lambda\text{ers}.\text{apply}_1c\ checklimiters \]
\[ \text{checklimit} = \lambda c(l, a)rs.(\text{limit } \in a) \rightarrow \text{new } c \text{ True } rs, \text{new } c \text{ False } rs \]

As indicated in the footnote in section 2.4.3 we could make \text{LIMIT} and \text{CONST} deep for environments. This is achieved by replacing the ordinary value \( r \) obtained from opening an environment \( E.E \) by \( r^\prime a' \) where \( a' \) is the access of the denotable value, and \( r^\prime \) is defined as follows:

\[
\begin{align*}
    r^\prime &= \lambda a^\prime l.(rl = \bot) \rightarrow \bot, r_1(rl) \\
    r_1 &= \lambda e.(\#e = 0) \rightarrow \langle \rangle, r_2(e \downarrow 1)$r_1(e\uparrow 1) \\
    r_1 &= \lambda (l, a).\langle [(l, a \cup a')] \rangle
\end{align*}
\]

This is not enough to avoid \text{THISENV} to make a copy of the entire environment, which has to be avoided by explicitly representing and checking for access information together with \( r \), i.e. replace \( r \) by \( (r, a) \). Note also that the making of functions and continuations inside an opened limited environment will also make a copy of the environment, and should be restricted. It is also a problem with processes remaining inside the opened limited environment, as they will have access to it until they leave it. In any case we cannot avoid individual denotable values to escape from a limited environment, but they will only escape as limited denotable values if the modifications indicated above are done.

### 5.2.12 Equality, subset and assignment

**Location equality, ==**

Location equality is simple, we just have to compare the location of each pair of arguments. It raises an error if the arguments have different length, otherwise it returns a multiple value of booleans.

\[ E|E_1 == E_2|c = \lambda\text{ers}. E[E_1|c_1] \text{ers} \]
\[ c_1 = \lambda\text{ers}_1.E[E_2|c_2] \text{ers}_2 \]
\[ c_2 = \lambda\text{ers}_2.(\#e_1 = \#e_2) \rightarrow \text{apply}_2c_2\text{eqloc }e_1e_2rs, \text{new } c_3\text{ers}_1 \]
\[ c_3 = \lambda\text{ers}_1.\text{new } c_4\text{ers}_2 \]
\[ c_4 = \lambda\text{ers}_2.\text{differrange }c(e_1, e_2)rs \]
\[ \text{eqloc} = \lambda c(l_1, a_1)(l_2, a_2)rs.(l_1 = l_2) \rightarrow \text{new } c \text{ True } rs, \text{new } c \text{ False } rs \]

93
Value equality, =

Value equality is complex for deep and recursive structures. It contains an occur check, and only succeed if the values have the same aliases. The test is recursive on the actual values. The central algorithm is `requal` which used the following information:

- \( c_t \) which is the continuation used if the values are equal, this initially makes a new `TRUE` denotable value, but can be changed to a continuation raising a limited error if a limited denotable value was encountered.

- \( c_f \) which is the continuation used if the values are different. Here this makes a new `FALSE` denotable value.

- \( L_1 \) and \( L_2 \) which are two list of locations, access information and actual values which are still to be compared. When two new environments with equal names and multiplicity of denotable values (or references) are encountered their denotable values are expanded and added to these lists.

- \( S_1, S_2 \) and \( k \) are used to keep track of aliases and circular structures, i.e. when the value of two locations are compared, they are recorded in \( S_1 \) and \( S_2 \) with the same unique integer \( k \). If any of these locations are encountered later without the other also being encountered the values are different. If both are encountered again the alias structure is equal, and the these two components of the \( L \) lists are equal.

Value equality returns a multiple value of booleans. The equality test raises an error if the arguments are of different length, or if they are equal except for limited values (which cannot be compared). An example:

\[
\{\text{INT } a; \ \text{LIMIT INT } b; \ \text{BOOL } c\} = \\
\{\text{INT } a; \ \text{LIMIT INT } b; \ \text{TRUE } c\} \quad /* \text{FALSE} */
\]

\[
\{\text{INT } a; \ \text{LIMIT INT } b; \ \text{BOOL } c\} = \\
\{\text{INT } a; \ \text{LIMIT INT } b; \ \text{BOOL } c\} \quad /* \text{limited} */
\]

Thus the algorithm attempts to find inequalities even if it has encountered limited denotable values.

\[
E[E_1 = E_2]c = \lambda rs. E[E_1]c_1rs \\
c_1 = \lambda e_1rs. E[E_2]c_2rs \\
c_2 = \lambda e_2rs.(\#e_1 = \#e_2) \to \text{apply}_2c \text{ equal } e_1e_2rs, \text{ new } c_3e_1rs \\
c_3 = \lambda e_1rs. \text{new } c_4e_2rs \\
c_4 = \lambda e_2rs. \text{diffrange } c(e'_1, e'_2)rs \\
equal = \lambda c_d. drs. \text{expand } m_o m_c c_5 d_1 d_2 \\
c_5 = \lambda l_1 l_2. \text{requal } c_3 c_f l_1 l_2(\lambda i.\{})(\lambda i.\{})(0 \\
requal = \lambda c_f l_1 l_2 S_l S_k. (L_2 = \{} \to c_t S_1 S_2 k, \text{ access } (L_1 \downarrow 1)(L_2 \downarrow 1) \\
c_f = \lambda v. \text{new } c_a \text{ False} \\
c_t = \lambda S_1 S_2 k. \text{new } c_a \text{ True} \\
94
access = \lambda((l_1, a_1, v_1)(l_2, a_2, v_2)).
        (limit \in a_1) \rightarrow \text{requal}(\lambda S_1 S_2 k.\text{limited} v_0(d_1)rs)cf(L_1 \uparrow 1)(L_2 \uparrow 1)S_1 S_2 k,
        (limit \in a_2) \rightarrow \text{requal}(\lambda S_1 S_2 k.\text{limited} v_0(d_2)rs)cf(L_1 \uparrow 1)(L_2 \uparrow 1)S_1 S_2 k,
        (a_1 \neq a_2) \rightarrow cf\ False,\ alias)

alias = (S_1 l_1 \neq S_2 l_2) \rightarrow cf\ False,
        (((S_1 l_1 = S_2 l_2) \land (S_2 l_2 \neq \{\}))) \rightarrow \text{requal} cf(L_1 \uparrow 1)(L_2 \uparrow 1)S_1 S_2 k,
        \text{value}(S_1[l_1/(S_1 l_1 \cup \{k\})](S_2[l_2/(S_2 l_2 \cup \{k\}]]))(k + 1)(\text{list} v_1)(\text{list} v_2)

value = \lambda S_1 S_2 k u_1 u_2.(\#u_1 = \#u_2) \rightarrow apply a cf cf equal u_1 u_2 S_1 S_2 k, cf\ False

c_7 = \lambda S_1 S_2 k.\text{requal} cf(L_1 \uparrow 1)(L_2 \uparrow 1)S_1 S_2 k

equal = \lambda c_1 c_f w_1 w_2 S_1 S_2 k.(w_1 \in \text{Env} \land w_2 \in \text{Env}) \rightarrow
        ((\#w_1 = \#w_2) \rightarrow apply b c_8 c_f env w_1 w_2, cf\ False), c_{12}

c_8 = \lambda l_1 l_2.\text{apply}_c.(\text{expand} m_1 m_0)\text{c}_9 l_1 l_2

c_9 = \lambda l_1 l_2.\text{requal} c_f l_1 l_2 S_1 S_2 k

eval = \lambda c_1 c_f(l_1, e_1)(l_2, e_2).l_1 = l_2 \land \#e_1 = \#e_2) \rightarrow c_1 e_1 e_2, cf\ False

c_{12} = (w_1 \in (DVals \setminus \text{Nil}) \land w_2 \in (DVals \setminus \text{Nil})) \rightarrow \text{dval} c_1 c_f w_1 w_2, c_{14}

dval = (w_1 = w_2) \rightarrow c_8 e_1 e_2, cf\ False

c_{14} = (w_1 = w_2) \rightarrow c_1 S_1 S_2 k, cf\ False

apply a = \lambda c_1 c_f u_1 u_2 S_1 S_2 k.(\#u_1 = 0) \rightarrow c_1 S_1 S_2 k,
        cf c_f(u_1 \downarrow 1)(u_2 \downarrow 1)S_1 S_2 k

c_1 = \lambda S_1 S_2 k.\text{apply}_a c_f f(u_1 \uparrow 1)(u_2 \uparrow 1)S_1 S_2 k

apply b = \lambda c_1 c_f u_1 u_2.(\#u_1 = 0) \rightarrow c_1(\emptyset), cf c_f(u_1 \downarrow 1)(u_1 \downarrow 1)

c_1 = \lambda l_1 l_2.\text{apply}_b c_f c_f(u_1 \uparrow 1)(u_2 \uparrow 1)

c_2 = \lambda l_1^f l_2^f.\text{cf}(l_1^f S_1^f)(l_2^f S_2^f)

apply c = \lambda c f u_1 u_2.(\#u_2 = 0) \rightarrow c(\emptyset), cf c(u_1 \downarrow 1)(u_2 \downarrow 1)

c_1 = \lambda l_1^f l_2^f.\text{apply}_c c_f(u_1 \uparrow 1)(u_2 \uparrow 1)

c_2 = \lambda l_1^f l_2^f.\text{cf}(l_1^f S_1^f)(l_2^f S_2^f)

expand = \lambda m_1 m_2 c_e(l_1, a_1)(l_2, a_2).c_e(l_1, a_1, m_1 s_1 l_1)(l_2, a_2, m_2 s l_2)

Deep inequality is defined in terms of equality, i.e.:

E[E_1 <> E_2]c = \lambda ers.E[\text{NOT} (E_1 = E_2)]c ers

Subset, LEQ

The algorithm of subset test is similar to that for equality except that equality (\_\_\_) is replaced by subset (\_\_\_), and the main algorithm \textit{requal} uses the same parameters. Note however that the algorithm is complicated by union values, where \textit{subset} checks that each component of the left hand side (u_1) is the subset of one component on the right hand side (u_2). We have here used the assumption that we can get a list of the union components from a union value. Note also that we use the false continuation \textit{cf} to assure that all components of u_2 are tried. The \textit{apply}_{a,b,c} functions are the same as those defined for equality.
\[ E[1 \text{ LEQ } E2]c = \lambda e. E[E1]|c|e_1rs \\
c_1 = \lambda r.E[E2]|c|\text{ers} \\
c_2 = \lambda e_2rs.(\#e_1 = \#e_2) \rightarrow \text{apply}_e \text{cleval}_e|e_2rs, \text{new } c|e_1rs \\
c_3 = \lambda e_1rs.\text{new } c|e_2rs \\
c_4 = \lambda e_2rs.\text{diffrange}_c(e_1', e_2')rs \\
\]

\[ \text{leqval} = \lambda c_2d_1d_2rs.\text{expand } m|t\text{mc}|c|d_1d_2 \\
c_5 = \lambda l_2.\text{reqlval}_c|c|l_2(\lambda i.\{\}) (\lambda i.\{\}) 0 \\
\]

\[ \text{reqlval} = \lambda c_3c_4c_5c_6c_7. (L_2 = \{\}) \rightarrow c_8c_9c_10c_11s_k, \text{access } (L_1 \downarrow 1)(L_2 \downarrow 1) \\
c_7 = \lambda v.(v = \text{False}) \rightarrow \text{new } c_a|\text{False}s_v, v \\
c_8 = \lambda S_1S_2k.\text{new } c_a|\text{True}s \\
\]

\[ \text{access} = \lambda (w_1, v_1, t_1). (L_2, v_2, t_2). ((\text{limit } \leq a_1) \rightarrow c_f(\text{limited } c_a|d_1)rs, \\
((\text{limit } \leq a_2) \rightarrow c_f(\text{limited } c_a|d_2)rs, \\
((a_1 \neq a_2) \rightarrow c_f \text{False, alias}) \\
\]

\[ \text{alias} = (S_1 \uparrow l \uparrow L_2) \rightarrow c_f \text{False,} \\
(((S_1 \uparrow l \uparrow L_2) \uparrow (S_2 \uparrow k)) \rightarrow \text{reqlval } c_3c_4c_5c_6l_1c_10c_11S_1S_2k, \\
\text{value } (S_1l_1[S_1 \uparrow l \cup \{k\}]) \rightarrow \text{reqlval } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
((k + 1)(\text{list } v_1)(\text{list } v_2)) \\
\]

\[ \text{value} = \lambda S_1S_2k.\text{value } (u_1 = \text{Void} \lor (u_2 = \text{Div}v)) \rightarrow \\
\text{reqlval } c_3c_4c_5c_6l_1c_10c_11S_1S_2k, \\
\text{subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\]

\[ \text{subset} = \lambda c_3c_4c_5c_6l_1c_10c_11S_1S_2k. (u_1 = \{\}) \rightarrow \text{reqlval } c_3c_4c_5c_6l_1c_10c_11S_1S_2k, \\
(u_2 = \{\}) \rightarrow c_f \text{False, subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\]

\[ \text{subset1} = \lambda S_1S_2k.\text{subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k, \\
\text{subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\]

\[ \text{subset2} = \lambda S_1S_2k.\text{subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k, \\
\text{subset } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\]

\[ \text{env} = \lambda c_3c_4c_5c_6l_1c_10c_11S_1S_2k. (\#w_1 = 0) \rightarrow c_8c_9c_10c_11S_1S_2k, \text{value } (\#w_1 > \#w_2) \rightarrow c_f \text{False,} \\
\text{idenv } (w_1 \downarrow 1)(w_2 \downarrow 1)(w_1 \uparrow 1)(w_2 \uparrow 1) \\
\]

\[ \text{idenv} = \lambda (l_1, e_1)[l_2, e_2].w_1 \downarrow 1.l_1 = l_2 \rightarrow \\
\text{value } (\#e_1 \geq \#e_2) \rightarrow \text{apply}_e \text{idenv}_v(\text{expand } m|m_4) e_1 | e_2, c_f \text{False,} \\
((l_1 > l_2) \rightarrow c_f \text{False, idenv } (l_1, e_1) (w_2 \downarrow 1)(w_1 \uparrow 1)) \\
\]

\[ \text{idenv1} = \lambda l_2. \text{apply}_e \text{idenv}_v(\text{reqlval } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\text{env} = \lambda S_1S_2k.\text{env } c_3c_4c_5c_6l_1c_10c_11S_1S_2k \\
\text{dval} = \lambda c_3c_4c_5c_6l_1c_10c_11S_1S_2k. (\#e_1 \geq \#e_2) \rightarrow \\
\text{apply}_e \text{idenv}_v(\text{expand } m|m_4) e_1 | e_2, c_f \text{False} \\
\]

The algorithm also attempts to avoid raising the limited exception, e.g.:

\[
(\{\text{LIMIT INT a}\} \cup \{\text{INT b}\}) \text{ LEQ } (\{\text{INT b}\}) \quad /* \text{ TRUE }*/
\]
\[
(\{\text{LIMIT INT a}\} \cup \{\text{INT b}\}) \text{ LEQ } (\{\text{INT a}\}) \quad /* \text{ limited e1 }*/
\]
\[
(\{\text{LIMIT INT a}\} \text{ LEQ } (\{\text{INT b}\}) \quad /* \text{ FALSE }*/
\]
\[
(\{\text{INT a}; \text{ LIMIT INT b}\} \text{ y LEQ } (\{\text{INT a}\}) \quad /* \text{ TRUE }*/
\]
\[
x := y \quad /* \text{ OK }*/
\]

LEQ is the same check as is done before assignment, i.e. the maximal value of the left hand side is checked against the actual value of the right hand side, and for pointers and environments the maximal values are compared. To get other tests,
TYPE and VAL must be used, e.g.

\[
\begin{align*}
\text{VAL} \ a \ &\ \text{LEQ} \ \text{VAL} \ b \\
\text{TYPE} \ a \ &\ \text{LEQ} \ \text{TYPE} \ b \\
(\text{VAL} \ a \ &\ \text{LEQ} \ \text{VAL} \ b) \ &\ \text{AND} \ (\text{VAL} \ b \ &\ \text{LEQ} \ \text{VAL} \ a) \\
/* \ a = b */ 
\end{align*}
\]

The last one is not the same with regards to the limited error, the first one will raise an error in any case, \(a = b\) will only raise an error when they are otherwise equal.

**Assignment, := and =:**

Assignment is shallow, but there is a deep subset check on the maximal value of the destination and the actual value of the source. The destination is also checked not to be a constant denotable value.

The assignment is not parallel, i.e. \(a := 4; \ a,b := 3,a\) will give \(b\) the value 3 regardless of \(a\)’s previous value.

\[
\begin{align*}
E[E_1 := E_2]c &= \lambda e.rs.E[E_1]c_1e.rs \\
c_1 &= \lambda e_1.rs.E[E_2]c_3e.rs \\
c_2 &= \lambda e_2.rs.(&e_1 \neq e_2) \rightarrow \text{apply}_2 c \ \text{assign} \ e_2e_1rs, \ \text{new} \ c_3e_1rs \\
c_3 &= \lambda e_1.rs.\text{new} \ c_4e_2rs \\
c_4 &= \lambda e_2.rs.\text{differ}c(e_1', e_2')rs \\
\text{assign} &= \lambda c_a(l_1, a_1)d_2rs.\text{(const } e a_1) \rightarrow \text{constant} \ c_a((l_1, a_1))rs, \\
&\text{expand} \ m_vm_vc_5(l_1, \{\})d_2 \\
c_5 &= \lambda l_1l_2.\text{reval}c_4c_7l_2(\lambda l.\{\})\lambda l.\{\})0 \\
\text{reval} &= \lambda c_5c_7L_1L_2S_1S_2k.(L = \{\}) \rightarrow c_5S_1S_2k, \ \text{access} \ (L_1 \downarrow 1)(L_2 \downarrow 1) \\
c_n &= \lambda v.\text{notsubtype}c_a((l_1, a_1), d_2)rs \\
c_t &= \lambda S_1S_2k.c_a(l_1, a_1)r((\lambda (m, n).(m[l_1/(m_vsl_1, m_vsl_2]), n))s)
\end{align*}
\]

The rest is as for LEQ.

\[
\begin{align*}
E[E_1 := E_2]c &= \lambda e.rs.E[E_1]c_1e.rs \\
c_1 &= \lambda e_1.rs.E[E_2]c_3e.rs \\
c_2 &= \lambda e_2.rs.(&e_1 \neq e_2) \rightarrow \text{apply}_2 c \ \text{assign} \ e_1e_2rs, \ \text{new} \ c_3e_1rs \\
\end{align*}
\]

The rest is as for \(E := E\).

**5.2.13 Copy, value and type**

Copy, value and type are a bit complicated by the need to update the static environment of function values. This is done by delaying the copying of functions till the end, and then the static environment can be fixed, as all copied locations are collected in \(S\). The treatment of continuations are not correct, as it would require the explicit representation of the \(e\) and \(r\) stacks which are introduced in section 5.4. This extension is not difficult, just a bit messy.
Copy, COPY E

\[ E | \text{COPY} | E | c = \lambda r.E | E | c_1 r s \]
\[ c_1 = \lambda r.E | E | c_1 r s \]

*copy*  
\[ \lambda c_a(l, a) Scpp(m, n_i, n_f). (\text{limit } \in a) \rightarrow \text{limited } c_0 d_0 r s_0, \]
\[ ((S l \neq \{\}) \rightarrow c_a((S l, a)) Scpp(m, n_i, n_f)), \]
\[ \text{apply}_d c_2 (\text{dup } c_3) | (\text{list}(m_v(m, n_i, n_f))))(S | l/n_i) \]
\[ c_3 = \lambda Scpp.s.\text{apply}_d c_3 (\text{dup } c_3) | (\text{list}(m_v s l)) Scpp.s \]
\[ c_4 = \lambda Scpp.s.\text{c}_c(n_i, a) Scpp.s \]
\[ c_5 = \lambda Scpp.s.\text{apply}_c(c_6) | (\text{copy } Scpp.s, \]
\[ (v \in \text{Env}) \rightarrow \text{apply}_c(c_6) | (v Scpp.s, \text{dup}_1) \]
\[ \text{dup}_1 = (v \in \text{Fun} \lor v \in \text{Cont}) \rightarrow \text{c}(\lambda S.\text{update } c_v c_v v S v s s) s, c_v Scpp.s \]
\[ \text{env} = \lambda c(Scpp.s.\text{apply}_c c_4 \text{ copy } Scpp.s, \]
\[ c_7 = \lambda Scpp.s.\text{c}(l, e) Scpp.s \]
\[ \text{apply}_d = \lambda c_f Scpp.s. (\#v = 1) \rightarrow f c(v \downarrow 1) Scpp.s, f c_5(v \downarrow 1) Scpp.s \]
\[ c_8 = \lambda Scpp.s.\text{apply}_c(c_9) | (v Scpp.s) \]
\[ \text{apply}_c = \lambda c_f Scpp.s. (\#e = 0) \rightarrow c() Scpp.s, f c_6(e \downarrow 1) Scpp.s \]
\[ c_9 = \lambda Scpp.s.\text{apply}_c(c_9) | (v Scpp.s) \]
\[ c_7 = \lambda Scpp.s.\text{c}(e \delta e) Scpp.s \]
\[ \text{update} = \lambda c_f c_v(f, r, n_f') S x s.\text{apply}_c c_5 \text{change}_r S V s s \]
\[ c_8 = \lambda r S x (m, n_i, n_f). c c_p(f, r, n_f) S (m, n_i, n_f + 1) \]
\[ \text{change}_1 = \lambda c(l, e) S x s.\text{apply}_c c_0 \text{ change}_2 e S x s \]
\[ c_9 = \lambda e S x s.\text{c}(l, e) S x s \]
\[ \text{change}_2 = \lambda c(l, a) S x s. (S l = \{\}) \rightarrow c(l, a) S x s, c(S l, a) S x s \]

Note that the list of seen values \( S \) is propagated through all the multiple values. This means that \( \text{COPY} (\text{INT } a, a) \) will retain the aliases. This is similar for value, type and unification.

**Value**

Value is very similar to copy, and only the difference is given.

\[ E | \text{VAL} | E | c = \lambda r.E | E | c_1 r s \]

...
Type

Type is also similar to the two previous ones.

\[ E[\text{TYPE } E]c = \lambda rs.E[E]c_1rs \]

\[ \text{copy} = \lambda c_a(l, a)S(m, n_l, n_f).(\text{limit } \in a) \rightarrow \text{limited } c_0d_0rs_0, \]

\[ ((S_l \neq \{\}) \rightarrow c_6((S_l, a))S(m, n_l, n_f), \]

\[ \text{apply}_d c_2(\text{dup } c_6)((\text{list}(m_l(m, n_l, n_f)l))(S[l/n_l])) \]

\[ c_p(m[n_l/(\text{Void}, \text{Void}]), n_l + 1, n_f)) \]

\[ \ldots \]

5.2.14 Union and unification

Union

The union of two denotable values \(a\) and \(b\) is a new denotable value whose maximal value is the union of the maximal values of \(a\) and \(b\), and whose actual value is the union of the actual values of \(a\) and \(b\).

\[ E[E_1 \text{ UNION } E_2]c = \lambda rs.E[E_1]c_1rs \]

\[ c_1 = \lambda e_1rs.E[E_2]c_3rs \]

\[ c_2 = \lambda e_2rs.(\#e_1 = \#e_2) \rightarrow \text{apply}_2 e \text{ union } e_1e_2rs, \text{ new } c_3e_1rs \]

\[ c_3 = \lambda rs.\text{new } e_4e_2rs \]

\[ c_4 = \lambda e_2rs.\text{differrange } c(e_1, e_2)rs \]

\[ \text{union} = \lambda c((l_1, a_1), (l_2, a_2))rs.c_5(l_1, a_1)rs \]

\[ c_5 = \lambda (l_1, a_1)rs.(\text{limit } \in a_1) \rightarrow \text{limited } (c_6c_5)((l_1, a_1))rs,c_7(l_2, a_2)rs \]

\[ c_7 = \lambda (l_2, a_2)rs.(\text{limit } \in a_2) \rightarrow \text{limited } (c_6c_7)((l_2, a_2))rs,c_8(l_2, a_2)rs \]

\[ c_8 = \lambda (l_2, a_2)rs.\text{new } c_0(m_1s_1 \cup m_2s_2)rs \]

\[ c_0 = \lambda (l, a)r(m, n).c(l, a)r(m[l/(m_1(m, n)l), (m_0s_1 \cup m_0s_2))], n) \]

\[ c_e = \lambda cer_1sc(e \downarrow 1)rs \]

Grounded

Grounded checks if the actual values are singleton sets, returning either TRUE or FALSE. Note again that the ground check is shallow.

\[ E[\text{GROUNDED } E]c = \lambda rs.E[E]c_1rs \]

\[ c_1 = \lambda e_1rs_1.\text{apply}_1c \text{ ground } e_1rs \]

\[ \text{ground} = \lambda c(l, a)rs.(\text{limit } \in a) \rightarrow \text{limited } cdrs, \]

\[ (\text{set}(m_0s_1) = \text{False}) \rightarrow \text{new } c \text{ True } rs, \text{ new } c \text{ False } rs \]

\[ \text{set} = \lambda v.(\#(\text{list}v) = 1) \land (v \neq \text{Int}) \land (v \neq \text{Char}) \land (v \neq \text{Bool}) \land (v \neq \text{Diov}) \land (v \neq \text{Void}) \rightarrow \text{False}, \text{ True} \]
Voided

Voided checks if some component of the actual value is VOID. This can be used to check if unification has "failed". Note that we also here need the list of visited locations \( S \) to treat recursive structures correctly.

\[
\begin{align*}
E|\text{VOIDed} \ E|c &= \lambda c. E|E|\text{cers} \\
c_1 &= \lambda c_1. \text{apply}_2 c \text{ voiders} \\
\text{voided} &= \lambda c_2(l_0, a_0): \text{rs} \Rightarrow \text{reqvoid} c_1 c_f((l_0, m_{e sl_0}, a_0))(\lambda i.\{\})0 \\
c_f &= \lambda u. \text{new} c_a \text{ Truers} \\
c &= \lambda S k. \text{new} c_a \text{ Falsers}
\end{align*}
\]

\[
\begin{align*}
\text{reqvoid} &= \lambda c_1 c_f L S k.(L = \{\}) \Rightarrow c_f S k, \text{ access } (L \downarrow 1) \\
\text{access} &= \lambda (l, v, a).(\lim \in a) \Rightarrow \text{reqvoid} (\lambda S k. \lim \in c_a((l_0, a_0)) \text{rs}) c_f(L \uparrow 1) S k, \text{ alias} \\
\text{alias} &= (S l \neq \{\}) \Rightarrow \text{reqval} c_f c_1 c_f(L \uparrow 1) S k, \\
& \quad \text{value } (S l/((S l \uparrow \{\}) \text{rs}))(k + 1)(\text{list } (v_1)) \\
\text{value} &= \lambda S k'. u.(u \neq \text{Void}) \Rightarrow \text{apply}_a c_f c_1 c_f \text{ notvoid } u S k', c_f \text{ False} \\
c_7 &= \lambda S k. \text{reqvoid} c_f c_1 c_f(L \uparrow 1) S k \\
\text{notvoid} &= \lambda c_1 c_f w S k.(w \in \text{Env}) \Rightarrow \text{apply}_b c_8 c_f \text{ env } w, c_{12} \\
c_8 &= \lambda l. \text{reqvoid} c_f c_1 c_f(L \uparrow 1) S k
\end{align*}
\]

\[
\begin{align*}
\text{env} &= \lambda c_1 c_f(l, e). \text{makelist } m_{e c_{1 e}} \\
c_{12} &= (w \in \text{DVals}) \Rightarrow \text{dval } c_1 c_f w, c_i S k \\
\text{dval} &= \lambda c_1 c_f e. \text{makelist } m_{e c_{1 e}} \\
\text{apply}_a &= \lambda c_1 c_f f u S k.(\#u = 1) \Rightarrow f c_f c_1(u \downarrow 1) S k, f c_1 c_f(u \downarrow 1) S k \\
c_1 &= \lambda v. \text{apply}_a c_f c_1 c_f(u \downarrow 1) S k \\
\text{apply}_b &= \lambda c_1 c_f f u.(\#u = 1) \Rightarrow f c_1 c_f(u \downarrow 1), f c_1 c_f(u \downarrow 1) \\
c_1 &= \lambda l_1. \text{apply}_b c_2 c_f c_1 c_f(u \downarrow 1) \\
c_2 &= \lambda l_2. c_1 l_1 \langle l_2 \rangle \\
\text{makelist} &= \lambda m c_e.(\#e = 1) \Rightarrow \text{expand } c(e \downarrow 1), \text{makelist } c_1(e \uparrow 1) \\
c_1 &= \lambda l. \text{expand } c_2(e \downarrow 1) \\
c_2 &= \lambda l'. c_1 l' \langle l \rangle \\
\text{expand} &= \lambda m c_e(l, a). c_e \langle(l, m s l, a)\rangle
\end{align*}
\]

**Unification**

The algorithm for unification is quite complex. It treats recursive structures and set based unification. Also the stati environments of functions unified with D10UV is treated correctly by matching it up with the new location if any of the denotable values took part in the unification. Continuations are as for copy not treated properly, but again it is just a messy detail, which can be easily fixed.

The algorithm first construct the greatest lower bound in the store. This construction gives a lot of garbage as more and more aliases are discovered. This garbage is reclaimed by reconstructing recon the store afterwards, only using one location for each alias set.
For unification we need the following standard operations on sets of locations with a distinguished representative element:

- $A \in \text{Alias} = ((\text{Loc})^* \times \text{Loc} \times \text{Access})^*$, which is a set of alias sets and their representative location and access value.

- $\text{union} : \text{Alias} \rightarrow (\text{Loc} \times \text{Access}) \rightarrow \text{Loc} \rightarrow \text{Loc} \rightarrow \text{Alias}$, which takes a set of alias sets and two existing representative locations, and a new location and accesses, and makes a new set of alias sets where the new location is representative of the union of the two sets previously represented by the existing locations.

- $\text{find} : \text{Alias} \rightarrow \text{Loc} \rightarrow (\text{Loc} \times \text{Access})$, which finds the representative location and access for a location.

Note that the representative location is a new location, i.e. not one belonging to one of the merged sets. This is to avoid inplace unification.

\[
\begin{align*}
E|E_1 \sim E_2|c &= \lambda ers. E[E_1]c_1ers \\
c_1 &= \lambda c_1 c_2. E[E_2]c_2ers \\
c_2 &= \lambda c_2 r.(m_0, n_0). (\#c_1 = \#c_2) \rightarrow \text{apply}_m \text{recon}_t \text{unify} e_1 e_2 m_0 (m_0, n_0), \\
c_3 &= \lambda c_3 r. \text{new} c_4 c_2 r s \\
c_4 &= \lambda c_4 r. \text{distrib} r. c(e_1^t, e_2^t) rs \\
\text{recon}_t &= \lambda e As. \text{apply}_u (\lambda e S.s.\text{cers}recon e(\lambda t. \{ \})) s_0 \\
\text{unify} &= \lambda c_4 (l_1, a_1)(l_2, a_2). A(m, n, n_f). \text{access} \\
\text{access} &= (\text{limit} \in a_1) \rightarrow \text{limited} c_0 d_1 r s_0, \\
& ((\text{limit} \in a_2) \rightarrow \text{limited} c_0 d_1 r s_0), \text{alias} (\text{find} A_l1)(\text{find} A_l2) \\
\text{alias} &= \lambda (l_1, a_1)(l_2, a_2). (l_1 = l_2) \rightarrow \\
& c_6 (n_1, a_1 \cup a_2) (\text{union} A(l_1 l_2(n_1, a_1 \cup a_2))(m[\text{ml} l_1], n_1 + 1, n_f), \\
& \text{apply}_v c_6 (f c_6 (\text{list}(m_1 s_1))) (\text{list}(m_1 s_1)) \\
& (\text{union} A(l_1 l_2(n_1, a_1 \cup a_2))(m[\text{ml}(\text{Void}, \text{Void})], n_1 + 1, n_f)) \\
\text{f} &= \lambda c_7 (c v_1 A). \text{apply}_u c_7 (\lambda e c v_1) A s \\
c_7 &= \lambda c_8 c_9 (\text{find} A_n) A s \\
c_9 &= \lambda c_9 (c v_9 A) (\lambda (l, a). c A(m[\text{ml} l_1 \cup (v, \text{Void}]), n_1, n_f)) (\text{find} A_n) \\
c_10 &= \lambda c_10 c_11 (\lambda (l, a). c A(m[\text{ml} l_1 \cup (\text{Void}, v)], n_1, n_f)) (\text{find} A_n) \\
\text{unif} &= \lambda c_11 c_12 w_1 w_2 A s_2 w_1 \in \text{Env} \land w_2 \in \text{Env} \rightarrow \text{env}(c c) w_1 w_2 A s, c_12 \\
c_12 &= (w_1 \in (\text{DVals} \setminus \text{Nil}) \land w_2 \in (\text{DVals} \setminus \text{Nil}) \rightarrow ((\#w_1 > \#w_2) \\
& \rightarrow \text{apply}_m (c_14 (w_1 \#(\#w_2))) \text{unify} (\text{drop} w_1(\#(\#w_2))) w_2 A s, \\
& \text{apply}_m (c_14 (w_2 \#(\#w_2))) \text{unify} w_1 (\text{drop} w_2(\#(\#w_1))) A s) \\
& c_14 (w_1 \sim w_2) A s \\
\text{recon} &= \lambda c_1 (l, a) S(m, n). c_1 (A l) \\
c_1 &= \lambda (l, a). (\text{limit} \in a) \rightarrow \text{limited} c_0 d_0 r s_0, ((\text{Sl} \neq \{ \}) \rightarrow c_4 ((\text{Sl}, a)) S(m, n), \\
& \text{apply}_d c_2 (\text{dup} c_1) (\text{list}(m s_0 l))(S[l/n]((m[n](\text{Void}, \text{Void})), n + 1)) \\
c_2 &= \lambda S s. \text{apply}_d c_3 (\text{dup} c_e) (\text{list}(m s_0 l)) S s \\
c_3 &= \lambda S s. c_4 (a, n) S s \\
& \text{--- rest as for copy, with copy replaced by recon ---}
\end{align*}
\]
\begin{align*}
env & = \lambda c_1 l_2 A s. (\#l_1 = 0) \rightarrow c_l A s, ((\#l_2 = 0) \rightarrow c_l A s, env_1(l_1 \downarrow 1)(l_2 \uparrow 1)) \\
env_1 & = \lambda (l_1, e_1) (l_2, e_2). (l_1 = l_2) \rightarrow \left((\#e_1 > \#e_2) \rightarrow \right. \\
& \left. \text{apply}_m (env_3 l_1 \uparrow 1 (e_1 \uparrow (\#e_2))) \text{unify} (\text{drop} e_1 (\#e_2)) e_2 A s, \text{apply}_m \\
& \text{unify}_1 (e_2 \uparrow (\#e_1)) \text{unify} e_1 (\text{drop} e_2 (\#e_1)) A s, env_2 \\
env_2 & = \lambda (l_1 > l_2) \rightarrow env_3 (l_2 \downarrow 1) e_2 A s, env_3 l_1 \uparrow 1 (e_1) A s \\
env_3 & = \lambda n_1 n_2 e r e u A s. \text{env} env_4(l_1 \downarrow n_1)(l_2 \uparrow n_2) A s \\
env_4 & = \lambda u A s. c((l_1, e_2 u) \uparrow 1) u A s \\
apply_m & = \lambda c f l_1 l_2 A s. (\#l_1 = 1) \rightarrow f c(l_1 \downarrow 1)(l_2 \uparrow 1) A s, f c_1(l_1 \downarrow 1)(l_2 \uparrow 1) A s \\
c_1 & = \lambda e_1 A s. \text{apply}_m c_2 f(l_1 \uparrow 1)(l_2 \uparrow 1) A s \\
c_2 & = \lambda e_2 A s. c(e_1 \uparrow e_2) A s \\
apply_u & = \lambda c f l A s. (\#l = 0) \rightarrow c A s, f c_1(l \uparrow 1) A s \\
c_1 & = \lambda A s. \text{apply}_u c_f (l \uparrow 1) A s
\end{align*}

**Interaction between unification, subset, copy and unions**

Unions can be seen as a set of non union constructable values, where duplicates and subsets are eliminated. Unification of two union sets, \( u \) and \( v \) with values \( u_i \) and \( v_j \) is then the union of \( u_i \sim v_j \) for all \( i \) and \( j \), where again we remove duplicates and subsets.

A union \( u \) is a subset of a union \( v \) if all \( u_i \) is a subset of some \( v_j \).

A copy of a union copies each component.

**5.2.15 Sequential control flow**

This is all very simple and standard, the only interesting case is the use of multiple denotable values as the condition for **IF** and **WHILE**.

\[
\begin{align*}
E[\text{IF} \ E_1 \ \text{THEN} \ E_2 \ \text{ELSE} \ E_3 \ \text{FI}]c & = \lambda e_0 r s. E[E_1]c e_0 r s \\
& \quad e_1 = \lambda e_1 r s. E[E_2]c e_1 r s \\
& \quad branch = \lambda d r s. \text{isBool} e_2 d r s \\
& \quad c_2 = \lambda r s. (v = \text{True}) \rightarrow E[E_2]c e_0 r s, E[E_3]c e_0 r s
\end{align*}
\]
\[ E[\textbf{WHILE} \ E_1 \ \textbf{DO} \ E_2 \ \textbf{OD}]c_0 = \lambda e_0 r_0 s_0. E[E_1 | c_1 e_0 r_0 s_0] \]
\[ c_1 = \lambda e_1 s. \text{apply}_1 c_0 \ \text{loop} \ e_1 s \]
\[ \text{loop} = \lambda c_1 d r s. \text{isBool} \ c_2 d r s \]
\[ c_2 = \lambda v r s. (v = \text{True}) \rightarrow \text{oncemore} \ e_0 r_0 s, c_0 e_0 r_0 s \]
\[ \text{oncemore} = \lambda c e_0 r_0 s. E[E_2 | c_n e_0 r_0 s] \]
\[ c_n = \lambda e_2 r_2 s_2. E[\textbf{WHILE} \ E_1 \ \textbf{DO} \ E_2 \ \textbf{OD}]c_n e_2 r_2 s_2 \]

The use of multiple values in the loop condition is dubious, but here are at least some examples:

```clojure
fun -> n; fib := 0;
   n >>
      while -> n; n => 0, n > 2 do
         if n = 0 then fib := fib + 1 fi
         n - 1
      od;
   fib
nuf fibonacci;

fun -> n; /* {VOID} nullenv; {nullenv left; nullenv right} */
   node := 0;
   n >>
      while if -> n = nullenv then node := node + 1; false
         else n.left <> nullenv, n.right <> nullenv fi
         do
            if n.left <> nullenv then n.left
                else n.right fi
            od
   node
nuf countnodes;
```

Thus there is a connection between a multiple value in a loop condition and non linear recursive functions.

### 5.2.16 Functions and continuations

Functions and continuations are also quite simple, note however that both application of functions and continuations can incorporate multiple denotable values, which will be applied in sequence, and the results will be combined into a multiple denotable value.
\[ E[FUN E NUF]c = \lambda e \tau (m, n_i, n_f). \text{new } c(E, r, n_f) \tau (m, n_i, n_f + 1) \]

\[ E[E_1(E_2)]c = \lambda e_1 r s . E[E_1]c_1 r s \]
\[ c_1 = \lambda e_1 r s . E[E_2]c_2 r s \]
\[ c_2 = \lambda e_2 r s . \text{apply } c \text{ call } e_1 r s \]
\[ \text{call} = \lambda cdrs . \text{isFun } c_3 drs \]
\[ c_3 = \lambda (E_2, r_f, n) . E[E_2]c_r e_2 r_f s \]
\[ c_r = \lambda erds . cers \]

\[ E[E_1//E_2]c = \lambda e_1 r s . E[E_1]c_1 r s \]
\[ c_1 = \lambda e_1 r s . E[E_2]c_2 r s \]
\[ c_2 = \lambda e_2 r s . \text{apply } c \text{ go } e_1 r s \]
\[ \text{go} = \lambda cdrs . \text{isCont } c_3 drs \]
\[ c_3 = \lambda (c, r_c, n) . ce_2 r_c s \]

\[ E[E_1////E_2]c = \lambda e_1 r s . E[E_1]c_1 r s \]
\[ c_1 = \lambda e_1 r s . E[E_2]c_2 r s \]
\[ c_2 = \lambda e_2 r s . \text{apply } c \text{ goce } e_1 r s \]
\[ \text{goce} = \lambda cdrs . \text{isCont } c_3 drs \]
\[ c_3 = \lambda e_1 r s . \text{new } (e_4 \nu) (c, r, n_f) \tau (m, n_i, n_f + 1) \]
\[ c_4 = \lambda (c, r_c, n) e_4 r s . c(e_5 \xi e_2) r_c s \]

\[ E[\text{STATENV} E]c = \lambda e_1 r s . E[E_1]c_1 r s \]
\[ c_1 = \lambda e_1 r s . \text{apply } c \text{ statenvs} \]
\[ \text{statenv} = \lambda c(l, a) r s . \text{limit } e \in a \rightarrow \text{limited } c((l, a)) r s, \]
\[ ((m_v \in \text{ Contv } \cup \text{ Fun }) \rightarrow \text{ new } c((\lambda (x, r, n) . E(m_v \nu))) r s), \]
\[ \text{notFUNorCONT } c((l, a)) r s \]

The forms where \( E_2 \) is replaced by nothing indicating a VOID parameter will give the following changes (only the alternative for \( E_1(E_2) \) is given):

\[ E[E_1()]c = \lambda e_1 r s . E[E_1]c_1 r s \]
\[ c_1 = \lambda e_1 r s . \text{apply } c \text{ call } e_1 r s \]
\[ \text{call} = \lambda cdrs . \text{isFun } c_3 drs \]
\[ c_3 = \lambda (E_2, r_f, n) . E[E_2]c_r e_2 r_f s \]
\[ c_r = \lambda erds . cers \]

### 5.2.17 Input and output

The specification of input and output is given in a different, more verbose form. This is because it contains a lot of simple checks which are tedious to write out in
full, and that the parsing and unparsing of Nel is not defined.

INPUT(E1)(E)

- Evaluate E1 to ei, if VOID then use MYPID.std.in for ei.
- Evaluate E, making e.
- Check if ei is a pointer to a multiple value of characters (text). If it is 
  continue, if it is NIL schedule another process (let this one try with regular 
  intervals, i.e. polling), and if it is neither return the error nottext (a syntax 
  error).
- Parse the text to find the first expression, and make the syntax tree Es. If a 
  syntax error occurred return an error message.
- Make a reference to the rest of the text, i.e. REF rest(n+1)(text), and 
  assign that to MYPID.std.in or ei.
- Make a new continuation λkers.E[Es|k1e{}ş, where k₁ = λer₁.s.kers.

INCHAR(E)

Similar to INPUT(E)(), but returns only the first character. The denotable value 
is returned, thus no copy is made.

OUTPUT(E1)(E)

- Evaluate E1 to eo, if VOID then use MYPID.std.out for eo.
- Evaluate E, making e.
- Unparse the value of e making a multiple character value, Etext blanks and 
  CR are inserted to make it print nice.
- Check if eo is a reference to a text, if it is make:

  ei := REF (DEREF ei, Etext);

OUTCHAR(E1)(E)

Similar to OUTPUT(E1)(E).
5.3 Parallelism

This section contains the semantic domains to model the parallel constructs in Nel together with the semantic equations for the parallel constructs. The semantic model is perhaps even more operational for the parallel construct than for the sequential ones, and the definition is close to an actual implementation of the run-time system for Nel. The most important change would be to redefine the interleaving function $P$ from making all interleavings to one arbitrary.

5.3.1 The semantic domains

The store is extended with information about which process is running, which are active, how many semaphores are used, what are the status of the semaphores, how many processes are used, and what are the status of each process.

The status of a semaphore records how many are waiting for the semaphore, who is waiting, and a denotable value which holds the external information about this semaphore.

The status of a process contains the last denotable value, environment and continuation of the process together with status information, if it is executing a limited continuation or function, which processes are waiting for it, how many clones it has, and an external denotable value which gives information about the process.

$$\text{State} = \text{State}_{\text{seq}} \times \text{Running} \times \text{Actives}$$
$$\times \text{NoSem} \times \text{Sem} \times \text{NoPr} \times \text{Map}$$

$$p \in \text{Actives} = \text{Pid}^*$$
$$N \in \text{NoPr} = \text{Int}$$
$$N_r \in \text{NoSem} = \text{Int}$$
$$i \in \text{Running} = \text{Int}$$
$$M_s \in \text{Sem} = \text{Int} \rightarrow (\text{Int} \times \text{Waitlist} \times \text{ExSem})$$
$$W \in \text{Waitlist} = \text{Pid}^*$$
$$E \in \text{ExSem} = \text{DVal}$$
$$M \in \text{Map} = \text{Pid} \rightarrow (\text{Cont} \times \text{DVals} \times \text{Env} \times \text{Status}$$
$$\times \text{Exec} \times \text{Wait} \times \text{Clone} \times \text{ExPid})$$
$$S \in \text{Status} = \{\text{active, halted, finished}\} \cup (\{\text{process, semaphore}\} \times \text{Int}^+)$$
$$i \in \text{Pid} = \text{Int}$$
$$C \in \text{Clone} = \text{Pid}^*$$
$$X \in \text{Exec} = \text{Id} + \text{Void}$$
$$E \in \text{ExPid} = \text{DVal}$$

Auxilliary functions

The following auxilliary functions are used:
\(rm\) removes the pid \(i\) from the list of pids \(i^*\) and is defined as
\[
rm = \lambda i.i.l.(i_i(i_i \downarrow 1)) \rightarrow (i_i i^1), (i_i i^1(rm (i_i i^1)i))
\]

\(add\) adds a list of integers
\[
add = \lambda l.(\#l = 0) \rightarrow 0, (l \downarrow 1) + (add(l \uparrow 1))
\]

\(where\) takes an integer and a list of integers and returns hwere in the list this integer occurs:
\[
where = \lambda li.where_1li1 \quad where_1 = \lambda liw.(l \downarrow 1 = i) \rightarrow w, \text{where}_1(l \uparrow 1)i(w + 1)
\]

The following auxiliary functions to check if a denotable value is an original process identifier or semaphore are defined:

\[
\begin{align*}
\text{checkpid} &= \lambda cc_f(l, a)rs.(m, sl \in Env) \rightarrow c_1(m, sl), \text{notPID} c_f(l, a)rs \\
c_1 &= \lambda r.((r \text{ pid}) \neq \bot) \land ((r \text{ pidnr}) \neq \bot) \rightarrow c_2(r \text{ pid})(r \text{ pidnr}), \\
& \quad \text{notPID} c_f(l, a)rs \\
c_2 &= \lambda (l, a, f)(l, n, a).((m, sl) \in \text{Fun} \land (m, sl) \in \text{Int}) \rightarrow \\
& \quad c_3(m, sl)(m, sl)ss, \text{notPID} c_f(l, a)rs \\
c_3 &= \lambda (f_1, r_1, I_1)i((m, n), i, p, N, M, N, M).c_4(M) \\
c_4 &= \lambda (k', e', r', S, X, C, (l', a')).c_5(m, sl) \\
c_5 &= \lambda r'.c_6(r \text{ pid}) \\
c_6 &= \lambda (l, a).((m, sl) = (f_1, r_1, I_1)) \rightarrow cis, \text{notPID} c_f(l, a)rs \\
\text{checksem} &= \lambda cc_f(l, a)rs.(m, sl \in Env) \rightarrow c_1(m, sl), \text{notSEM} c_f(l, a)rs \\
c_1 &= \lambda r.((r \text{ sema}) \neq \bot) \land ((r \text{ semnr}) \neq \bot) \rightarrow c_2(r \text{ sema})(r \text{ semnr}), \\
& \quad \text{notSEM} c_f(l, a)rs \\
c_2 &= \lambda (l, n, a).((m, sl) \in \text{Fun} \land (m, sl) \in \text{Int}) \rightarrow \\
& \quad c_3(m, sl)(m, sl)ss, \text{notSEM} c_f(l, a)rs \\
c_3 &= \lambda (f_1, r_1, I_1)i((m, n), i, p, N, M, N, M).c_4(M) \\
c_4 &= \lambda (l, s, W_s(l, a))c_5(m, sl) \\
c_5 &= \lambda r'.c_6(r \text{ sema}) \\
c_6 &= \lambda (l, a).((m, sl) = (f_1, r_1, I_1)) \rightarrow cis, \text{notSEM} c_f(l, a)rs
\end{align*}
\]

5.3.2 The interleaving model

The interleaving model makes all possible interleavings of the processes, thus making the final answer a powerset of states, i.e.:

\[
P : \text{Exp} \rightarrow \text{Cont} \rightarrow \text{DVals} \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{Power(State)}
\]

\text{Power(\text{State})} is represented as a list of states

 Atomic actions

Each operation in Nel is an atomic action. Thus interrupts (switching process) can only take place between operations. To specify this we change the semantic equations as follows:

107
$$\mathbb{P}[E]k = \lambda r((m,n),i,p,N_s,M_s,N,M).
\forall (m,n),p,N, (\text{upd } M(Mi)(\lambda ers.E_s[E]kers)er)p
\quad \forall (m,n),p,N, \text{ord}(p = \emptyset) \rightarrow ((m,n),p,N_s,M_s,N,M),
\quad (l = \emptyset) \rightarrow \emptyset,
\quad \text{do}((l+1)(((m,n),p,N_s,M_s,N,M)\text{ord}(p,g,N_s,M_s,N,M)(l+1))))
\quad \text{do}_1 = \lambda k,e,r,S,X,W,C,E \text{.ker}((m,n),i,p,N_s,M_s,N,M)
\quad \text{upd} = \lambda M(k',e',r',S,X,W,C,E) \text{ker}.M[i/(k,e,r,S,X,W,C,E)]$$

$E_S$ will here apply one atomic operation, i.e. $E_S[E_1 \text{ op } E_2]$ is $\mathbb{P}[E_1]$ and $\mathbb{P}[E_2]$ combined with "E[op]".

To get this to work smoothly we make all processes which have done all updating of the map, $M$, return $-1$, which is a dummy number.

A simple example:

$$\mathbb{P}[\text{INT}][k] = \lambda r((m,n),i,p,N,M).\forall (m,n),p,N_s,M_s,N,
(\text{upd } M(Mi)(\lambda ers.E_s[\text{INT}kers]er)p
= \lambda r((m,n),i,p,N_s,M_s,N,M).\forall (m,n),p,N_s,M_s,N,
(\text{upd } M(Mi)(\lambda ers.\text{ers.new c Int rskers}er)p
= \lambda r((m,n),i,p,N_s,M_s,N,M).\forall (m,n),p,N_s,M_s,N,
(\text{upd } M(Mi)(\lambda ers.\text{ers.new k Int rs}er)p$$

In the beginning

The initial state and continuation is given by the following equations, i.e. the evaluation of $E$ starts as process 0 in an empty environment preceeded by VOID, and is ended by STOP.

$$\mathbb{P}[\text{VOID};E;\text{STOP}](\lambda ers. \bot) \bot (\lambda i. \bot)((((\lambda i. \bot),1,0),0,(\lambda i. \bot),1,(\lambda i. \bot))$$

Here we could replace VOID and STOP by any prelude and termination code.

### 5.3.3 Changes to the Sequential model

This section describes some minor changes which have to be worked into the semantic equations for the sequential part. This is not done in full detail, but does not represent any difficulties.

**Executing a limited function or continuation**

For every function or continuation which is called, $X$ is updated with the identifier of this function. The continuation resets the environment and $X$. This is used to indicate if the process is executing a limited continuation or function. Note that the calling of a limited function or continuation is not transitive, i.e. calling a non limited function from a limited one sets $X$ to Void. Processes executing a limited
function or continuation can only be halted by LIMITHALT, and the state will not be disclosed when the process is waiting.

Note that processes which do not execute a limited function or continuation and are waiting for either other processes or semaphores get their external process identifiers updated. This might not be a fortunate design, and perhaps the process identifier only should be updated by explicit HALT and SUSPEND. This will be trivial to change.

**Turn of access and constant checks for process 0**

This is not specified, but might be a way to provide some kind of superuser privilege. One problem in the present specification is that a process executing a diverging limited function which no one else has access to cannot be stopped. This is avoided by a superuser.

**Exceptions**

The function raise defined to search for the exception handler, which returned ⊥ when it did not find a handler is changed to suspend the process, i.e. ⊥ is replaced by suspend (e+1)rs

The continuation to reraise the exception is dropped when the process is suspended.

### 5.3.4 The parallel constructs

We can now specify the parallel constructs similarly to the sequential ones, the difference being that they affect the part of the state modelling the processes. This section specifies the different constructs. The specifications are a bit messy due to a lot of details, but an informal description is given before each one.

**Stop**

This terminates the process, i.e. removes it from the active list, and updates its status. It also starts all processes which were waiting on the termination of this process. Note that a process can wait for more processes, and only when the last process is terminated will the waiting process resume.

\[
\begin{align*}
E[\textbf{STOP}] & = \lambda e((m, n), i, p, N_s, M_s, N, M).c_1(M) \\
\lambda & = \lambda (k', e', r', S, X, W, C, (l, a)) . \\
\text{apply} & = (c_2((m, n), l)\text{status}))((\text{waitlist True i})W rs \\
\lambda & = \lambda (l_{\text{stat}}, o_{\text{stat}})W r((m, n), i, p, N_s, M_s, N, M).c e r \\
\text{(m, l_{\text{stat}}/m, n, l_{\text{stat}}, v_{\text{finished}})}, n), -1, (\text{rm ip), N_s, M_s, N, M[i/(\bot, \bot, \bot, finished, X, \bot, C, (l, a))})
\end{align*}
\]
\[
\text{waitlist} = \lambda v_i c_i r((m, n), i, p, N_s, M_s, N, M). w_1(M_i) \\
\text{wl}_1 = \lambda (k_p, e_p, r_p, (T, i_t), X, W, C, (l, a)). (\text{add}(i_t[i_h/0]) = 0) \rightarrow \\
\text{start}(e_p \downarrow (\text{where} i_h i_t))((m_u(m, n)l_h)\text{cont}) \\
((m_r(m, n)l_h)\text{dval})((m_u(m, n)l_h)\text{env})((m_u(m, n)l_h)\text{status}), \\
\text{wl}_2(e_p \downarrow (\text{where} i_h i_t)) \\
\text{wl}_2 = \lambda (l_p, a_p). c_i r((m[l_p/(m_l(m, n)l_p, v)]), n, i, p, N_s, M_s, N, \\
M[i_p/(k_e, e_r, (T, (i_t[i_h/0]))), X, W, C, (l, a)])) \\
\text{start} = \lambda (l_p, a_p)(l_c, a_c)(l_e, a_e)(l_r, a_r)(l_{\text{stat}}, a_{\text{stat}}). c_i r \\
((m[l_p/(m_l(m, n)l_p, v)])(l_c/(m_l(m, n)l_c)), \text{Void})[[l_e/(m_l(m, n)l_e), \text{Void}]] \\
[l_r/(m_l(m, n)l_r), \text{Void}][l_{\text{stat}}/(m_l(m, n)l_{\text{stat}}, \text{active})], n, \\
i, (p_s(i_p)), M_s, N_s, N, M[i_p/(k_p, e_p, r_p, \text{active}, X, W, C, (l, a))]) \\
\]

Note that \text{waitlist} are used by several semantic equations, and is parametrized with a value, \(v\), indicating the reason why the process ceased waiting for a process or semaphore together with which process or semaphore it ceased waiting for, \(i\). The apparently unmotivated \(T\) appearing in the parameter list of \text{wl}_1 is just a dummy variable which can be either \text{semaphore} or \text{process}.

\textbf{Mypid}

Mypid makes the \textit{ExPId} \((E)\) value of the running process, \(i\), the current denotable value of this process.

\[
E[\text{MYPID}]c = \lambda e r((m, n), i, p, N_s, M_s, N, M). c_1(M_i) \\
c_1 = \lambda (k', e', r', S, X, W, C, E). c E r((m, n), i, p, N_s, M_s, \\
(N+1), M[i/(k, E', r, S, X, W, (C\text{s}(N)), E)][N/(k, E, r, S, X, \langle, \langle, E')])
\]

\textbf{Fork}

Fork clones the process doing fork. \textit{newpid} is an allocation of a new external process identifier (environment) for the clone.

\[
E[\text{FORK}]c = \lambda e r((m, n), i, p, N_s, M_s, N, M). c_1(M_i)(\text{newpid} N(m, n)) \\
c_1 = \lambda (k', e', r', S, X, W, C, E)(m, n) E'. k E' r((m, n), -1, (p s(N)), N_s, M_s, \\
(N+1), M[i/(k, E', r, S, X, W, (C\text{s}(N)), E)][N/(k, E, r, S, X, \langle, \langle, E')])
\]

110
\[ \text{newpid} = \lambda N(m, n_i, n_f). (m[n_i/(\text{Diov, Void})][n_i + 1/(\text{Diov, Void})] \]
\[ \quad [n_i + 2/(\text{Diov, Void})] \]
\[ \quad [n_i + 3/(N, N)][n_i + 4/(\text{Diov, get in})][n_i + 5/(\text{Diov, get out})] \]
\[ \quad [n_i + 6/(\text{Diov, v_active})][n_i + 7/(\text{Diov, v_active})] \]
\[ \quad [n_i + 8/\langle\langle \text{in}, (n_i + 4, \{\})\rangle, \text{out}, (n_i + 5, \{\}), \text{err}, (n_i + 6, \{\})\rangle\rangle, \]
\[ \quad \langle\langle \text{in}, (n_i + 4, \{\})\rangle, \text{out}, (n_i + 5, \{\}), \text{err}, (n_i + 6, \{\})\rangle\rangle, \]
\[ \quad [n_i + 9/\langle\langle \langle \text{cons}, (n_i + 2, \{\})\rangle, \text{std}, (n_i + 3, \{\}), \text{cont}, (n_i + 7, \{\})\rangle, \text{pid}, (n_i + 9, \{\})\rangle\rangle, \]
\[ \quad [\langle\langle \text{cons}, (n_i + 2, \{\})\rangle, \text{std}, (n_i + 3, \{\}), \text{cont}, (n_i + 7, \{\})\rangle, \text{pid}, (n_i + 9, \{\})\rangle\rangle, n_i + 11, n_f + 1 \]
\[ n_i + 10, \{\} \]
\[ \text{get} = \lambda l. (\lambda (k, e, r, S, X, W, C, (l, a)). \]
\[ \quad (\lambda (l, a). m_i(m, n) l ((m_i(m, n))((m_i(m, n)))(\text{std})) l)(M_i) \]

**PS**

PS uses the list of clones, which is a list of process identifiers, to index into the map of processes, and selects the process identifiers which are returned.

\[ \text{E}[\text{PS}(E)]c = \lambda c. \text{E}[\text{E}[\text{E}[\text{c}_{\text{E}}]]] \]
\[ c_1 = \lambda c. \text{E}[\text{E}[\text{E}[\text{apply_1 c_pers}}] \]
\[ ps = \lambda k(l, a) r s. \text{checkpid c}_2 c_2 c_2(l, a)r s \]
\[ c_2 = \lambda p(m, n, i, p, N, M, N, M). \text{c}_3(M_i p) \]
\[ c_3 = \lambda (k_p, e_p, r_p, S_p, X_p, W_p, C_p, E_p). \text{apply_1 get}_E_c_2(c(s) p) r s \]
\[ \text{get}_E_c = \lambda c. ((\lambda (k, e, r, S, X, W, C, E). E)(M_i s)) r ((m, n), i, p, N_s, M_s, N, M) \]

**Wait**

Wait makes the process wait on the termination of a finite number of processes.

\[ \text{E}[\text{WAIT(E)}]c = \lambda c. \text{E}[\text{E}[\text{E}[\text{c}_{\text{E}}]]] \]
\[ c_1 = \lambda c. \text{E}[\text{E}[\text{E}[\text{apply_1 c_wait ers}}] \]
\[ c_2 = \lambda i_r c_4 c_4 (m, n, i, p, N_s, M_s, N, M). c_5(\text{split i_r})(M_i) \]
\[ c_3 = \lambda (i_p, e_r)(k', e', r', S, X, W, C, (l, a)). ((\text{add i}_p = 0) \rightarrow \]
\[ \text{c}_4(r ((m, n), i, p, N, M, N, M), \]
\[ ((X = \text{Void}) \rightarrow \]
\[ c_4((m_i(m, n)) l \text{cont}) ((m_i(m, n)) l \text{dval}) ((m_i(m, n)) l \text{env}) (m, n), \]
\[ c_5((m_i(m, n)) l \text{status}) (m, n)) \]
\[ c_4 = \lambda (l, c, a_r)(l, c, a_r)(l, r, M, n). c_5((m_i(m, n)) l \text{status}) \]
\[ (m_i/l_i(m_i(m, n))) l c) c_4, (m_i/l_i(m_i(m, n))) l c) c_4, (l, (m_i/l_i(m, n)) l r, n), n) \]
\[ c_5 = \lambda (l, a, a_r)(m, n). c_6((m_i/l_i(m, n)) l \text{status} \text{vwaiting proc}) (n), n) \]
\[ (l, (m_i/l_i(m, n)) l r, n), n) \]
\[
\begin{align*}
\text{split} &= \lambda l.l = \langle \rangle \to (\langle \rangle, \langle \rangle), ((\text{fst}(l \downarrow 1) \bowtie \text{split}(l \uparrow 1)), (\text{snd}(l \downarrow 1) \bowtie \text{split}(l \uparrow 1))) \\
\text{fst} &= \lambda (a, b). a \\
\text{snd} &= \lambda (a, b). b \\
\text{wait} &= \lambda k(l, a) r. \text{checkpid} c_2 c l a r s \\
   c_2 &= \lambda i_p((m, n), i, p, N_s, M_s, N, M). c_3(M_i_p) \\
   c_3 &= \lambda (k_p, e_p, r_p, S_p, X_p, W_p, C_p, E_p). (S_p = \text{halted}) \to \\
   &\quad k(0, (n, \{\}))(m[n/\{\text{Bool, False}\}], n + 1), i, p, N_s, M_s, N, M, \\
   &\quad k(i_p, (n, \{\}))(m[n/\{\text{Bool, Void}\}], n + 1), i, p, N_s, M_s, N, \\
   &\quad M[i_p/k_p, e_p, r_p, S_p, X_p, W_p, (W_p[i](i)), C_p, E_p]) \\
\end{align*}
\]

Here \( n \) should actually have been converted to \( n_i, n_f \), and every occurrence of \( n \) should be replaced by \( n_i \) to get it correct.

**Waiting**

Waiting gives out a list of process number of those processes waiting for the termination of a process. Note that waiting does not return the process identifiers.

\[
\begin{align*}
E[\text{WAITING}(E)]c &= \lambda e r s. E[E(c) e r s] \\
   c_1 &= \lambda e r s. \text{apply}_1 c \text{ waiting} e r s \\
\text{waiting} &= \lambda k(l, a) r. \text{checkpid} c_2 c l a r s \\
   c_2 &= \lambda i_p((m, n), i, p, N_s, M_s, N, M). c_3(M_i_p) \\
   c_3 &= \lambda (k_p, e_p, r_p, S_p, X_p, W_p, C_p, E_p). (W_p = \langle \rangle) \to \text{new} c \text{ Void} r s, \\
   &\quad \text{apply}_1 \text{ new} c W_p r s \\
\end{align*}
\]

**Sem**

Sem allocates a new semaphore (\( \text{SemNo} \) and \( \text{SemMap} \)) which is initialized to 1, and an external semaphore identifier is allocated in the store.

\[
\begin{align*}
E[\text{SEM}]c &= \lambda e r ((m, n_i, n_f), i, p, N_s, M_s, N, M). c(n + 1, \{\text{const}\}) r \\
   &\quad ((m[n_i/(N_s, N_s)])[n_i + 1/((- >, \{\}, n_f)), (- >, \{\}, n_f)]) \\
   &\quad [n_i + 2/(\langle \text{semr}, ((n_i, \{\text{const}\})), (\text{semid}, ((n_i + 1, \{\text{const}\})))\rangle), \\
   &\quad (\langle \text{semr}, ((n_i, \{\text{const}\})), (\text{semid}, ((n_i + 1, \{\text{const}\})))\rangle), n_i + 3, n_f + 1), \\
   &\quad i, p, (N_s + 1), M_s[N_s/(1, \{\}, n + 1, \{\text{const}\})], N, M) \\
\end{align*}
\]

**P**

P lets the process wait for a finite number of semaphores, and lets it continue when it has acquired all.
\[ E[P(E)]c = \text{\texttt{lers.E[E]c}}_{\text{ers}} \]

\[ c_1 = \text{\texttt{lers.apply}}_1 \text{\texttt{c \_ serns}} \]

\[ c_2 = \lambda l.c r((m, n), i, p, N_s, M_s, N, M).c(\text{split } l_e)(M_i) \]

\[ c_3 = \lambda (i_s, e).c((k', e', r', S, X, W, C, (l, a)))(\text{add } i_s = 0) \rightarrow \]

\[ ce_r r((m, n), i, p, N_s, M_s, N, M), ((X = \text{Void}) \rightarrow \]

\[ ce((m_e(m, n)l)\text{cont})((m_e(m, n)l)\text{dval})((m_e(m, n)l)\text{env})(m, n), \]

\[ c_5((m_e(m, n)l)\text{status})(m, n)) \]

\[ c_4 = \lambda (l, a_0)(l_0, a)(l_0, a)(m, n).c_5((m_e(m, n)l)\text{status}) \]

\[ (m|l_0/(m_t(m, n)l_0, c)]|l_0/(m_t(m, n)l_0, e_0)|[l_0/(m_t(m, n)l_0, r_0)], n) \]

\[ c_5 = \lambda (l, a_0)(l_0, a)(m, n).c_5((m|l_0/(m_t(m, n)l_0, a_0)(m, n)), n), 1, \]

\[ (rm \pi), N_s, M_s, N, M[i/(c, e, r, (\text{semaphore, } i_s), X, W, C, (l, a)])] \]

\[ \text{\texttt{semp}} = \lambda k(l, a)rs.\text{checksem } c_2 c(l, a)rs \]

\[ c_2 = \lambda i_s((m, n), i, p, N_s, M_s, N, M).c_5(M_s i_s)(M_i) \]

\[ c_3 = \lambda (l_s, W_s, E_s)(k', e', r', S, X, W, C, E).((l_s > 0) \rightarrow \]

\[ k(0, (n, {}))r((m|n/\text{Bool}, n + 1, i, p, \]

\[ N_s, M_s[i_s/((l_s - 1), W_s, E_s)], N, M), \]

\[ k(i_s, (n, {}))r((m|n/\text{Void}, n + 1, i, p, \]

\[ N_s, M_s[i_s/((l_s - 1), (W_s \downarrow 1), E_s)], N, M)) \]

Here we should also have replaced \( n \) with \( n_t, n_f \), as for \texttt{WAIT}. \texttt{split} was defined from \texttt{WAIT}.

**V**

\[ V \]

V releases a finite number of semaphores, and resumes any process which is only waiting for this semaphore. Note that a process can wait for more semaphores simultaneously, and will not be resumed until it has required all semaphores.

\[ E[V(E)]c = \text{\texttt{lers.E[E]c}}_{\text{ers}} \]

\[ c_1 = \text{\texttt{lers.apply}}_1 \text{\texttt{c \_ serns}} \]

\[ \text{\texttt{semp}} = \lambda c(l, a)rs.\text{checksem } c_2 c(l, a)rs \]

\[ c_2 = \lambda i_s((m, n), i, p, N_s, M_s, N, M).c_5(M_s i_s) \]

\[ c_3 = \lambda (l_s, W_s, E_s)(W_s = \langle \rangle) \rightarrow \]

\[ c((l, a))r((m, n), i, p, N_s, M_s[i_s/((l_s + 1), W_s, E_s)], N, M), \]

\[ \text{\texttt{waitlist True i_0 c(W_s \downarrow 1)r((m, n), i, p, \]

\[ N_s, M_s[i_s/((l_s + 1), (W_s \uparrow 1), E_s)], N, M)) \]

**Semwait**

Semwait is similar to waiting, and returns a list of the process numbers of processes waiting for a semaphore.
\[ E[\text{SEMWAIT}(E)]c = \lambda e.s.E[e]c\text{ers} \]
\[ c_1 = \lambda e.rs.\text{apply}_1 c \text{ semwait} rs \]
\[ \text{semwait} = \lambda k(l, a)rs.\text{checksem} c_2c(l, a)rs \]
\[ c_2 = \lambda i_z((m, n), i, p, N_s, M_s, N, M).c_3(M_zi_z) \]
\[ c_3 = \lambda (I_z, W_z, E_z).(W_z = \langle \rangle) \rightarrow \text{new } c \text{ Void } rs, \]
\[ \text{apply}_1 \text{ new } cW_zrs \]

Suspend

Suspend has two forms, one with 4 ordinary parameters, and one making a continuation. As the arguments of suspend are assigned to the process environment they are checked not to be limited, avoiding a breach in the security. The actual values of the parameters given to suspend are also checked to be of the correct type.

\[ E[\text{SUSPEND}(E)]c = \lambda e.rs.E[e]\text{suspenders} \]
\[ \text{suspend} = \lambda e.(\#e = 4) \rightarrow c_1(e \downarrow 1)(e \downarrow 2)(e \downarrow 3)(e \downarrow 4)rs, \text{new } c\text{d, e}rs \]
\[ c_1 = \lambda (l, c, a_c), (l, c, a_c), (l, a_r), (l, a_r), (l, a_r), a_{\text{stat}}.r((m, n), i, p, N_s, M_s, N, M). \]
\[ \text{access} = (\text{limit } \in a_c) \rightarrow \text{limited } c((l, a_c))rs, \]
\[ ((\text{limit } \in a_c) \rightarrow \text{limited } c((l, a_c))rs, \]
\[ ((\text{limit } \in a_c) \rightarrow \text{limited } c((l, a_c))rs, \]
\[ ((\text{limit } \in a_{\text{stat}}) \rightarrow \text{limited } c((l, a_{\text{stat}}))rs, \]
\[ \text{typecheck} = \lambda v_c v_e v_r.(v_c \notin \text{Contv}) \rightarrow \text{notCONT } c((l, a))rs, \]
\[ ((v_c \notin \text{Dval}) \rightarrow \text{notPOINTER } c((l, a))rs, \]
\[ ((v_r \notin \text{Env}) \rightarrow \text{notENV } c((l, a))rs, \]
\[ c_4 = \lambda (k', e', r', S, X, W, C, (l, a)).c_5((m_0(m, n)l)\text{cont})((m_0(m, n)l)\text{dval}) \]
\[ ((m_0(m, n)l)\text{env})((m_0(m, n)l)\text{status}) \]
\[ (m_0(m, n)l)\text{stat} \]
\[ c_5 = \lambda (l, e_c)(l, e_c)(l, a_r)(l, a_r)(l_{\text{stat}}, a_{\text{stat}})v_{\text{stat}}.\text{apply}_1 k \text{ (waitlist False }i)r \]
\[ ((m_0l_c/m_0(m, n)l_c, v_c)[l_c/(m_0(m, n)l_c, v_c)[l_c/(m_0(m, n)l_r, v_r)] \]
\[ [l_{\text{stat}}/(m_0(m, n)l_{\text{stat}}, v_{\text{stat}})], n), -1,(\text{rpm }i), N_s, M_s, N, \]
\[ M[i/(1, 1, 1, 1, \text{halted}, X, \langle \rangle, C, (l, a))] \]
\[ c_{d1} = \lambda e.rs.\text{new } c_d4rs \]
\[ c_{d2} = \lambda e.rs.\text{differange }c(e', e_2)rs \]

Halt

Evaluate E, apply \textit{halt} to the list. Check that each denotable value is a process identifier. There are some cases to consider:
The process to be halted is active. This is handled in active. The halted process is removed from the list of active processes, and its process identifier is updated.

One of the processes which are to be halted is oneself. This is handled by the test in c4, which replaces the current process by -1.

The halted process is waiting for other processes or semaphores. This is handled in proc and sem. Here the process has to be removed from those waiting. Note that acquired semaphores are not released automatically.

The halted process has already finished. This is handled in c5.

\[ E[HALT(E)]c = \lambda e_s.E[E](\text{ers}) \]
\[ c_1 = \lambda e_s.\text{apply}_1 c \text{ halt ers} \]
\[ \text{halt} = \lambda k(l,a)rs.\text{checkpid} c_2 c(l,a)rs \]
\[ c_2 = \lambda i_h((m,n),i,p,N_s,M_s,N,M).c_3(M_i_h) \]
\[ c_3 = \lambda (k_h,e_h,r_h,S_h,X_h,W_h,C_h,(i_h,a_h)). \]
\[ (S_h \in \{\text{halted, finished}\}) \rightarrow \text{new k Falsers, } c_{3'} \]
\[ c_{3'} = (X_h \neq \text{ Void}) \land (i_h \neq i) \rightarrow \text{new k Void rs, new } c_4 \text{ True rs} \]
\[ c_4 = \lambda e_r((m,n),i,p,N_s,M_s,N,M).(S_h = \text{ active}) \rightarrow \]
\[ \text{active}_1((\lambda i_h.(i = i_h) \rightarrow -1, i)i_h), c_5 \]
\[ \text{active}_1 = \lambda i.\text{active}_2((m_u(m,n)l_h)\text{cont}) \]
\[ ((m_u(m,n)l_h)\text{dval})(m_u(m,n)l_h)\text{env})(m_u(m,n)l_h)\text{status} \]
\[ \text{active}_2 = \lambda (l_c,a_c),(l_e,a_e),(l_r,a_r),(l_{\text{stat}},a_{\text{stat}}),k_e,r \]
\[ ((m_{l_c}(m_u(m,n)l_c),k_h)||l_{/}(m_{l_e}(m_u(m,n)l_e),e_h)||l_{/}(m_{l_r}(m_u(m,n)l_r),r_h)) \]
\[ ||l_{/}(m_{l_{\text{stat}}}(m_u(m,n)l_{\text{stat}},\text{halted})),n),i((\text{rm p}i_h)^5W_h), N_s, M_s, M[i_h/(\perp, \perp, \perp, \text{ halted}, X_h, ()], C_h,(i_h,a_h))] \]
\[ c_5 = (S = \{\text{process, } i_p\}) \rightarrow \text{apply}_1 c_7 \text{ proc } i_p rs, c_6 \]
\[ \text{proc} = \lambda c_{i_p}(l_h,a_h)r((m,n),i,p,N_s,M_s,N,M).(i = 0) \rightarrow \]
\[ c(l_h,a_h)r((m,n),i,p,N_s,M_s,N,M), \text{proc}_1(M_i_p) \]
\[ \text{proc}_1 = \lambda (k_p,e_p,r_p,S_p,X_p,W_p,C_p,E_p).c(l_h,a_h)r((m_{l_h}(m_u(m,n)l_h),\text{False}),n),i,p, \]
\[ N_s, M_s, M_i_p/(k_p,e_p,r_p,S_p,X_p, (\text{rm w}i_h), C_p, E_p)) \]
\[ c_6 = (S = \{\text{semaphore, } i_s\}) \rightarrow \text{apply}_2 c_7 \text{ sem } i_s e_s rs \]
\[ \text{sem} = \lambda c_{i_s}(l_h,a_h)r((m,n),i,p,N_s,M_s,N,M).(i = 0) \rightarrow \]
\[ c(l_h,a_h)r((m,n),i,p,N_s,M_s,N,M), \text{sem}_1(M_i_s) \]
\[ \text{sem}_1 = \lambda (I_s,W_s,E_s).c(l_h,a_h)r((m_{l_h}(m_u(m,n)l_h),\text{False}),n),i,p, \]
\[ N_s, M_i_s/(I_s - 1), (\text{rm w}i_h), E_s)], N,M) \]
\[ c_7 = \lambda e_r((m,n),i,p,N_s,M_s,N,M). \]
\[ \text{apply}_1(c_8(m_u(m,n)l_h)\text{status})((\text{waitlist False } i_h) W_h rs) \]
\[ c_8 = \lambda (l_{\text{stat}},a_{\text{stat}})e_r((m,n),i,p,N_s,M_s,N,M).k_e,r \]
\[ ((m_{l_{\text{stat}}}(m_u(m,n)l_{\text{stat}},\text{halted})),n),i((\text{rm p}i_h), N_s, M_s, N \]
\[ M[i_h/(k_h,e_h,r_h,\text{ halted}, X_h, ()], C_h,(i_h,a_h)]) \]

Limithalt

Limithalt is similar to halt, only that it halts just one process, and checks if the denotable value given as its second argument corresponds to a non-limit access to
the function or continuation the process is executing. This way of halting a limited process is not enough to avoid processes going astray, they can just define and execute a non-terminating function which they only have access to. An alternative which gives process 0 unlimited capabilities is discussed in section 5.3.3. The main reason for restricting the halting of processes executing limited functions are to allow “non-interruptable” code to be written. Note that when a process is halted with LIMITHALT its process identifier will be updated as for ordinary HALT.

\[ E[LIMITHALT(E)]=\\]
\[ c_1=\lambda\text{ers.}(\#e=2)\rightarrow c_{\text{limithalt}}\text{ers},\text{ new } c_{d_4}\text{ers} \]
\[ c_{d_1}=\lambda e''\text{rs}.\text{ new } c_{d_2}\text{rs} \]
\[ c_{d_2}=\lambda e''\text{rs}.\text{ diffrange } c(e', e_2)\text{rs} \]

\[ c_{\text{limithalt}}=\lambda ((l, a),(l_f, a_f))\text{rs}.\text{ checkpid } c_2 c(l, a)\text{rs} \]
\[ c_2=\lambda l_h((m, n), i, p, N_s, M_s, N, M).c_3(Ml_h)(Mi) \]
\[ c_3=\lambda (k_h, c_h, r_h, S_h, X_h, W_h, C_h, (l_h, a_h))(k', e', r', S, X, W, C, E). \]
\[ (S_h \in \{\text{halted, finished}\}) \rightarrow \text{ new } c \text{ False rs, } c_3n(m_v(m, n)l_f) \]
\[ c_3n=\lambda (f, r, l).((X_h \neq Void) \land (i_h \neq i) \land (X_h \neq l) \rightarrow \text{ new } c \text{ Void rs, new } c_4 \text{ Truers} \]

— rest as for halt

Restart

Restart resumes a halted process, i.e. a process which is previously explicitly halted with either HALT or LIMITHALT. As SUSPEND checks its parameters before they are assigned to the constant denotable values in the process identifier, no check is required when the process is resumed.

\[ E[RESTART(E)]=\\]
\[ c_1=\lambda\text{ers.}\text{ apply}_1 c \text{ restarters} \]

\[ \text{restart}=\lambda k((l, a)\text{rs}.\text{ checkpid } c_2 c(l, a)\text{rs} \]
\[ c_2=\lambda l_p((m, n), i, p, N_s, M_s, N, M).c_3(Ml_p) \]
\[ c_3=\lambda (k_p, c_p, r_p, S_p, X_p, W_p, C_p, E_p).(S_p \neq \text{ halted}) \rightarrow \text{ new } c \text{ False rs, } \]
\[ \text{ start }((m_v(m, n)l_h)\text{cont})((m_v(m, n)l_h)\text{dval}) \]
\[ ((m_v(m, n)l_h)\text{env})((m_v(m, n)l_h)\text{status}) \]

\[ \text{ start}_1=\lambda v_v.v_r.\text{ new } c \text{ True } r \]
\[ ((m[l_e/(m_v(m, n)l_e), Void])\text{con}(l_e/(m_v(m, n)l_e), Void)] \]
\[ r_v=((m[l_e/(m_v(m, n)l_e), Void])\text{con}(l_e/(m_v(m, n)l_e), Void)]\text{active}, X_p, W_p, C_p, E_p)) \]

5.4 Explicit continuations

The semantics given so far are in the style of standard (not direct) semantics, where we have used the lexical binding (closure) of the \(\lambda\)-calculus to implicitly represent the run time stack \(e, r\) and \(c\) stacks. An alternative method is to pass these stacks around explicitly.
This section gives a sketch of the changes needed to bring the semantic equations on the form of explicit continuations.

The form of explicit continuations is most appropriate for the abstract semantics discussed in the conclusion, chapter 6, as it is easier to keep track of references and handle loops.

The explicit handling of continuations is also necessary to give an appropriate description of the debugging environment and improve the external representation of continuations.

Garbage collection is also trivially specified on explicit continuations, i.e. all locations not reachable from any active process is garbage. A location is reachable if it can be reached through the e or r stacks. During garbage collection we can also collect deadlock information on semaphores and processes.

5.4.1 The semantic domains

A continuation, \( Cont \), is now a function taking lists (stacks) of continuations, denotable values and environments, a state, and gives out a state. A continuation value, \( contv \), is these three stacks and an identifier.

- \( c \in Cont \rightarrow Cont^+ \rightarrow DVals^+ \rightarrow Env^+ \rightarrow State \rightarrow State \)
- \( Contv = Cont^+ \times Env^+ \times DVals^+ \times Id \)
- \( Fun = (Cont \rightarrow Cont) \times Env \times Id \)

5.4.2 The semantic function

The semantic function \( E \) defined in section 5.2.3 must be changed accordingly.

\[
E : Exp \rightarrow Cont^+ \rightarrow DVals^+ \rightarrow Env^+ \rightarrow State \rightarrow State
\]

We let \( e \in DVals^+, r \in Env^+ \) and \( c \in Cont^+ \). We could also rename \( DVals^+ \) to \( DVals_{ss} \).

5.4.3 Changes to the semantic equations

Here only some of the changes to the semantic equations are given as examples. The changes are quite simple, but the representation turns out to be quite messy, as it requires a lot of explicit manipulation of the three stacks.

Auxiliary functions

\( apply_z \) takes one or two arguments on the \( e \) stack, and the continuation to be applied is on the \( e \) stack.
apply₂ : Conts → DVals → Envs → State → State
apply₁ = λcers.(#(e ⊥ 1) = 0) → (c ⊥ 2)(c↑2)(e↑1)rs,
     ((e ⊥ 1)((c₁)↑sc)(((((e ⊥ 1) ⊥ 1)), (e ⊥ 1)↑1))↑(e↑1))↑rs
     c₁ = λcers.apply₁c((e ⊥ 2), ((e ⊥ 1)↑(e ⊥ 3)))↑(e↑3)rs

apply₂ = λcers.(#(e ⊥ 1) = 0) → (c ⊥ 2)(c↑2)(e↑2)rs,
     ((e ⊥ 1)((c₁)↑sc)((((((e ⊥ 1) ⊥ 1)), ((e ⊥ 2) ⊥ 1)), (e ⊥ 1)↑1), (e ⊥ 2)↑1))↑(e↑2)rs
     c₁ = λcers.apply₁c((e ⊥ 2), (e ⊥ 3), ((e ⊥ 1)↑(e ⊥ 4)))↑(e↑4)rs

new : Conts → (Val + DVals)⁺ → Envs → State → State
new = λc(((v)↑sc)r(m, ni, n₀).c ⊥ 1((c↑1)(((ni, {i})))↑sc)r(m₁, ni + 1, n₁)
     m₁ = λi.(i = ni) → (v, v), ml

limited : Conts → DVals → Envs → State → State
limited = λcers(m, ni, n₀). apply₁ new (((c₁)↑sc)
     (((((lers.raise e limited rs)[sc, e, r, n₀], (c, e, r, n₀ + 1), e, r)r(m, ni, n₀ + 2)
     c₁ = λcers.c₁((e ⊥ 1)↑sc↑limited)↑(e↑2)rs
     c₂ = λe↑rs.raise e limited↑e↑rs

5.4.4 Semantic equations

Here only the changes to functions and continuations are given.

E[FUN E NUF] = λcers(m, ni, n₀). new (((E, r, n₀))↑sc)r(m, ni, n₀ + 1)

E[E₁ // E₂] = λcers.E[E₁][(c₁)↑sc][(e ⊥ 1)↑sc]rs
     c₁ = λcers.E[E₂][(c₂)↑sc][(e ⊥ 2), (e ⊥ 1)]↑(e↑2)rs
     c₂ = λcers.apply₁((c₂)↑sc) callcc (((e ⊥ 2), (e ⊥ 1))↑(e↑3)rs
     cₚ = λcers.(e ⊥ 1)(e↑1)(((e ⊥ 1))↑(e↑2)rs
     callcc = λcers.isFun (((c₁)↑sc)cers
     c₃ = λucer(m, ni, n₀).
     new (((cₚ)↑sc)(((((c₀)↑sc, (e↑1), r, n₀)))↑sc)r(m, ni, n₀ + 1)
     c₄ = λ(E₁, r, n₀)cers.E[E₂][(c₀)↑sc][(e ⊥ 1)(e↑4)]↑(e↑4)↑(r↑)↑rs
     cₕ = λcers.(c ⊥ 1)(c↑1)(e↑1)↑s

E[E₁ // E₂] = λcers.E[E₁][(c₁)↑sc][(e ⊥ 1)↑sc]rs
     c₁ = λcers.E[E₂][(c₂)↑sc][(e ⊥ 2), (e ⊥ 1)]↑(e↑2)rs
     c₂ = λcers.apply₁((c₂)↑sc) go (((e ⊥ 2), (e ⊥ 1))↑(e↑3)rs
     go = λcers.isCont ((c₂)↑sc)cers
     c₃ = λ(c₀, e, r, n₀)cers.(c ⊥ 1)(c↑1)(((e ⊥ 3)↑sc)↑c=((r ⊥ 1)↑)↑rs

This is not in any respect difficult, it is just an ordinary messy translation from implicit to explicit stacks.

Concurrency

The changes to the concurrent model are minor.
5.4.5 Storing the run time stacks

We can represent the continuations even more operationally. Presently the $e$, $r$ and $c$ stacks are monolithic. It is also possible to split them up into linked pieces, where each piece is stored in a location [DEUTSCH90]. The advantage of this is discussed in connection with abstract interpretation in the conclusion. This corresponds to activation records in an ordinary compiler.

The changes needed to the semantic domains and equations are simple. $Loc$ is here the location where the next piece of the continuation is stored.

\[
\begin{align*}
c \in \text{Cont} & \quad = \quad \text{Cont}^+ \to DVals^+ \to Env^+ \to Loc \to State \to State \\
\text{Contv} & \quad = \quad \text{Cont}^* \times Env^+ \times DVals^+ \times Loc \times Id
\end{align*}
\]

\[E : \text{Exp} \to \text{Cont}^* \to DVals^+ \to Env^+ \to Loc^+ \to State \to State\]

The only non-trivial changes in the semantic equations appear in functions and loops. When calling a function the current $c$, $e \downarrow 1$, $r$ and $l$ are saved in a new location $l'$, and the return continuation retrieves the stored continuation. (This seems similar to making an explicit stored continuation on each call.) The handling of loops (\texttt{while}) is similar, where the continuation is saved on each entry to the loop, and restored on each exit.

Chapter 6

Conclusion

This chapter contains some indications for further work, and an assessment of what has been achieved.

The possibilities of further work based on the ideas in this work are many. Some of them are already developed to a certain extent, but they have not reached a form which makes it natural to include any definite results here.

The description of further work is divided into the following sections. Section 6.1 elaborates some promising methods to improve the efficiency of Nel programs. In section 6.2 the possibility of using Nel as an intermediate language for to define other languages is considered. The implementation of Nel and its support environment is discussed in section 6.3.

6.1 Efficiency

An implementation of a Nel interpreter based on the semantic definition given in chapter 5 will obviously not be very efficient. This is the disadvantage of general and orthogonal constructs; they have a high overhead also for simple cases. The sources of inefficiency can be illustrated with the following function:

\[
\text{FUN} \rightarrow:\text{subset}(0,34)\ b;\ b + 3\ \text{NUF}
\]

The following redundant computations are made every time the code is interpreted:

- The \text{subset} function (defined in section 2.5.2) is called with constant parameters. The name \text{subset} is also dynamically looked up in the static environment every time. The denotable value is also checked to be a function.

- 0, 34 and 3 are dynamically allocated locations which are superfluous, as they are always constants. The locations remain garbage until collect by the general garbage collector, even if they are obviously not accessible after the
call of the function. No other attempt on cleaning up garbage after leaving an environment is done either.

- \texttt{b} is inserted into the environment, and looked up dynamically even if it will always be the same. (We do not know which location, if we have not evaluated \texttt{subset}.)

- \texttt{b} and \texttt{3} are checked to be of the same length, and \texttt{3} is checked to contain an integer value.

- Before the addition \texttt{b + 3} is attempted a check for overflow is made.

- All denotable values are checked not to be limited.

These are computations regularly performed by a compiler once and for all, leaving as little code as possible to be interpreted at run-time.

In addition to improving the efficiency of the interpreted code, compilers also trap some semantic errors (often termed static semantics). This is problematic in Nel, as all "errors" are caught by exceptions. So the only action a compiler can do is to give a message like: "Here the exception so and so will be raised, and the process running the program will be terminated."

From this example the need for a tool to do partial evaluation of Nel programs should be evident, and can give considerable improvements in the performance. Before we go on describing some ideas on such a tool, some cautious remarks must be made:

- Nel has an explicit and open representation of the internal data structure of the running program, which can be accessed by halting (\texttt{HALT}) the process at arbitrary places in the code. This information, which is then available in the process identifier, will not be complete for a partially evaluated function. This can give rise to breaches in the security if the right precautions are not taken. I.e. the optimized code skips a limited check for a denotable value, but the denotable value is replaced by another, and the process is restarted.

- The definition of exceptions makes the assumption that checking something once is enough, dubious. E.g. checking that the parameter \texttt{b} from above is a ground integer in connection with checking if it is a subset of 0...34 is not enough. The problem is that raising an exception too early is not correct, and we can risk that the process is restarted, which leads to the same problem as mentioned in the previous point.

- There is also the general problem of concurrent processes which can invalidate the assumptions that the performance improvements were based on, if care is not taken.

The problem of access to the state of a process running an optimized function can be avoided by giving each process a mark similar to the \texttt{Exec} mark for running limited code, and only updating the process identifier for interpreted functions.
The first two problems are comparable to problems with optimization across the boundary between the application program and the operating system. I.e. the compiler can skip many dynamic checks if it knows that an application program always has allowed parameters, but the code must be protected against "patching". Leaving these cautious remarks we can now look at the possibilities of improving the sequential and not interrupted code.

6.1.1 The results of partial evaluation

Partial evaluation should work on function values and give the following results:

- A more efficiently interpreted representation.

- Warnings where exceptions will occur and where run-time checks have to be performed.

- Information about requirements on parameters, and information about results, in the function from the example above the relevant information is 0..34 \rightarrow 3..37.

- Use of global variables, both read and written. The analysis can also optimize the function based on the current value of these variables. The analysis is then invalid if these values have changed when the function is run. A simple form of this caching is present in Make.

- Information about expensive run-time computations, e.g. dynamic search for identifiers.

Note that the method can always give up, and fall back on interpretation of certain pieces of the code where the techniques are not good enough.

As the orthogonal constructs in Nel replaced the need for special languages to express e.g. versioning, module interconnection and linking, the partial evaluator should replace the evaluation of these languages, which are mostly trivial compared with the real algorithm. The current phases of system evaluation can be illustrated in figure 6.1.

Each phase has its own syntax for representing the program, and the translation to the next phase is usually fixed, i.e. the translation algorithm does not use any heuristics. The exception is optimizing compilers, which employ different techniques to produce better code. This means that the syntax describing the program in each phase must be carefully chosen to allow the fixed translation. A language with a possibly non-terminating compiler is of limited use. In the end, the concern of this whole chain is to get an efficient machine code representation.

One advantage of splitting the evaluation into phases is to break the problem into smaller, more manageable pieces, and then provide tools for handling the problems of each phase separately. The problem is that the user has to relate to
Multi version code
  \downarrow Version selection
Single version code with macros
  \downarrow Preprocessor
  \downarrow Source code
  \downarrow Compiler
  \downarrow Assembly code
  \downarrow Linker
  \downarrow Machine code

Figure 6.1: The evaluation process of a software system

all the different description languages, and know when to use what, and which are their limitations. This is similar to many specialized constructs, or few orthogonal ones.

There is an interesting interplay between optimization and consequences of a change. If the system does extensive optimization, then a small change can invalidate the entire derived system. Whereas any change can be directly incorporated into a totally interpreted system.

6.1.2 Abstract interpretation

The method most promising to obtain the results indicated in the previous section seems to be abstract interpretation [ABRAMSKY87]. Abstract interpretation has been developed for functional languages, where some of the same problems occurs as in Nel.

The idea of abstract interpretation can be illustrated by the figure 6.2 below.

The abstract interpretation of the function over the finite abstract domains is used to collect information about the evaluation the original function over the standard domains. This information is used to optimize the original function. The finiteness of the abstract domains ensures the termination of the abstract interpretation.

The problem here is obviously to find adequate abstract domains which are rich enough to give some useful information, and yet finite. This problem has not been solved for Nel.

There are two main kinds of information which need to be collected:
- Information about the use of parameters and global variables. This is a backward flow problem.
- Information about the possible values of objects. This is a forward flow problem.

The interaction between these two flow problems, which can not be solved separately is the main problem.

Another problem which is avoided in most ordinary work on code optimization [AHO86] is the interconnection between flow of control and data flow. I.e. the presence of first class functions and continuations makes the flow graph dependent on the data values.

The most promising works which could be relevant to Nel are [DEUTSCH90] and [SHIVERS88,90]. In [DEUTSCH90] a framework for analysing imperative languages is set up. It will probably solve most problems except that I can not see how it can be used to collect information about parameters. Another problem is the treatment of control flow, which I think will give rather inaccurate results and be computational expensive. Deutsch represents continuations as linked “activation records”, and makes infinite recursion finite by making a loop of activation records. This loop represents the unknown depth of recursion, and the evaluation explores all possible solutions. This process can be pictured is figure 6.3 below.

Figure 6.3: Making continuations finite

One inaccuracy of this approach is that the number of calls is disconnected from
the number of returns. It also searches for a global fixed point, as there is no provision for making local fixed points.

Shivers is working on abstract interpretation of Scheme, and his work also includes the problem of collecting information about parameters. The problem is again the treatment of infinite continuations which seems even less satisfactory than the one proposed by Deutsch.

As mentioned some work has been done on the abstract interpretation of Nel, but no conclusive results are obtained. I hope to be able to continue some of this work.

6.2 Nel as an intermediate language

Throughout the examples in chapter 3, in appendix E and in [KARLSSON88] indications of how different language constructs map into the primitive mechanisms provided in Nel were given. These examples illustrate that it is comparatively easy to map other languages into Nel. By coupling a parser generator to Nel, a facility for generating interpreters for languages could be envisaged. Coupled with a partial evaluator this could be able to produce good code. The justification for this statement is found in the design of most languages. The designers have made a trade-off between expressibility and efficiency, where the efficient constructs can, to a large extent, be evaluated statically. These static evaluations will of course be represented intermixed with the dynamic evaluations in mapping into Nel. But as the dynamic evaluation does not interfere with the static evaluation, the partial evaluator should be able to automatically recover that information.

The porting of specialized techniques developed to improve the compiled code for particular languages should be interesting in its own right.

Interfaces to other languages

Using Nel as an intermediate language will also make all the other languages and the software written in these languages available in the Nel environment. It also gives the possibility to use specialized languages for specific tasks, and to integrate code from the different language on a higher level than machine code. This is similar to the approach taken in the Poplog environment [SLOMAN83].

Generating environments

The language translator tool described in the beginning of this section is just the first step in the development of a language environment. The effective use of a language also requires a programming support environment.

The main problem here is to establish the connection from the intermediate representation back to the source language, which the user wants to relate to. This problem is evident for e.g. error messages and debugger structures. The versioning
method presented in 4.6 will also fail to give syntactically well defined representations in the source language for merging of changes.

6.3 Implementation and use

An interpreter for an initial version of Nel as presented in [KARLSSON88] was implemented in the summer of 1988 [BRATSBERG88]. It was then used as an example language for a 3rd year course in programming languages. None of the ideas developed in the meantime have been implemented. Based on the experience from 1988, the implementation of an interpreter from the semantic specifications is relatively straightforward. The main bulk of work will consist of the implementation of DEI, which is a rather drastic extension of the Emacs editor. The implementation language should obviously be Nel.

6.4 Assessment

Now it is time to look back, and see what has been achieved. Looking at the list of further research I am left with the feeling of not having achieved what was originally intended. But considering the ambition of this list that was probably not surprising.

Considering the loose requirements given in section 1.3, an attempt of summing up can be attempted.

One language and one environment

Nel and DEI are certainly the only language and environment in this work. If this is enough remains to be seen, but experience from Lisp and Smalltalk systems should indicate that.

Computational model

Nel is a parallel imperative language, thus the computational model supports multiple cooperating users in one invocation.

Multi user support

The support for multiple users is provided by DEI, and satisfies the requirements for a loose form of locking found to be most appropriate in software development environments.
Programming styles and paradigms

Nel provides primitives for any programming style, and if the tool for generating language environments is realized, it should be possible to use most programming languages in the Nel environment.

User interaction

The user interacts with only one language and tool (DEI). The language to customize the editor is also the same.

Persistence

All denotable values in Nel are persistent, and all values can be stored in denotable values, so this requirement is fulfilled.

Versioning

The versioning is integrated into Nel and DEI, and the concept of a separate workspaces with integration determined by the user is also achieved.

Naming

The general environment values in Nel provide the most flexible dynamic search for names. Environments as first class values coupled with the static scoping of functions and continuations does not make the optimizing of functions impossible.

Typing

Nel has a flexible type system, where the actual value of a static value can be determined as precisely as required through the maximal value. As both the maximal and actual values are properties of the persistent denotable values, Nel is type safe. Furthermore, the input and output functions give the possibility to convert between different values.

Information hiding

The possibilities of making a denotable value limited and constant solves some of the problems of information hiding. Furthermore names can be hidden by the ordinary scope rules.
Appendix A

The complete syntax

This is the complete syntax for the Nel language, the reference in the last column
refers to the section where the concepts are described. \[E\] means optional.

\[
E ::= \begin{array}{l}
\text{VOID} \mid N \mid C \mid \text{TRUE} \mid \text{FALSE} \\
\text{INT} \mid \text{CHAR} \mid \text{BOOL} \mid \text{DIOV} \\
E + E \mid E - E \mid E \ast E \mid E / E \mid E \mod E \\
E < E \mid E <= E \\
E \text{ AND } E \mid E \text{ OR } E \mid \text{NOT } E \\
E \text{ Idlist} \mid \text{UNDEF Idlist} \mid \text{BEGIN E END} \\
I \mid \text{ISDEF Idlist} \\
\text{REF } E \mid \text{CONT } E \mid \text{NIL} \\
E,E \mid E[E] \mid \text{LEN } E \\
\{ E \} \mid \text{E } \& \text{ E} \mid \text{E.E} \mid \text{THISENV} \\
\text{CONST } E \mid \text{LIMIT } E \mid \text{ISCONST } E \mid \text{ISLIMIT } E \\
E \text{ UNION } E \mid \text{WIDEN } E \mid \text{GROUNDED } E \\
E \sim E \mid \text{VOIDED } E \\
E = E \mid E == E \mid E \prec E \mid E \leq E \\
E ::= E \mid E :: E \\
\text{COPY } E \mid \text{VAL } E \mid \text{TYPE } E \\
\rightarrow \mid E;E \mid E :: E \mid \text{(E)} \\
\text{IF } E \text{ THEN } E \text{ ELSE } E \text{ FI} \mid \text{WHILE } E \text{ DO } E \text{ OD} \\
\text{FUN } E \text{ NUF } \mid E[//](//)[E]) \mid \text{STATENV } E \\
\text{MYPID} \mid \text{FORK} \mid \text{SUSPEND}(//E) \\
\text{HALT(E)} \mid \text{RESTART(E)} \\
\text{STOP} \mid \text{PS(E)} \mid \text{WAITING(E)} \\
\text{SEM} \mid P(E) \mid V(E) \mid \text{SEMWAIT(E)} \\
\text{INPUT(E)}(E) \mid \text{INCHAR(E)} \\
\text{OUTPUT(E)}(E) \mid \text{OUTCHAR(E)}(E)
\end{array}
\]

\[
\begin{align*}
\text{Idlist} & ::= I:\text{Idlist} \mid I \\
I & ::= \text{identifiers, no maximum length} \\
N & ::= \text{decimal numbers, only positive integers, max }2^{63}, -x \text{ is written } 0-x \\
C & ::= \text{characters, 'c', or } \text{\}}23
\end{align*}
\]

Reserved words can be written either with only capital or only small letters, thus
\text{MYPID} is the same as mypid, but different from Mypid.
Appendix B

What was left out

This appendix contains a short overview of features considered to be included in Nel, but which were eventually left out. Brief discussions of some of the design choices made are also included.

B.1 Real time and priority

All processes in Nel have the same priority, and their execution is interleaved in an arbitrary way. There are no real-time facilities. To include these features in the language would complicate the semantic model considerably, and bring little to the discussion, as these features belong more to real-time systems than to programming languages.

Priority could also be established explicitly by assuring that no more than one process (with the best priority) is ready at any time. The modelling of real-time features is not possible.

Real-time features would not work well with some of the complex operations like unification etc. which are atomic actions, and do not have time bounds.

B.2 Transaction facilities

Transactions, atomic procedures, backtracking and choice points are all constructs to allow a complex operation to be executed without interference, and with possibilities for undoing its changes without affecting the other processes, i.e. commit or abort. The Argus language [LISKOV83] is an example of the incorporation of these constructs into a programming language.

A choice point is a complete state of the execution which is saved, and which can later be reactivated. A syntactic construct for making a choice point could be ||, and to reset the execution to a choice point the El construct could be used. A choice point must save the complete state of all processes, not only the one making
the choice point.

```
IF (|| cp) <> VOID THEN
  /* try something */
  IF /* fail */ THEN cp| FI
ELSE
  /* do something else */
FI
```

|| cp binds the choice point to cp, and when it is bound it is different from VOID. When the choice point is used by cp|, the resulting value from || is VOID.

Choice points are semantically not very complex, but their effect in a multiuser environment with external devices is not well defined, i.e. the state of the user can not be saved. This is similar to optimistic concurrency control in databases.

Another approach is to define all “global” variables read after a choice point as having a read lock, and all written to have a write lock, and to release all these locks when the choice point is not longer possible to activate. This is similar to lazy locking which is used in databases, and semantically quite complex.

### B.3 Deep assignment

A deep assignment := and ==: for environments and pointers could have been included. The meaning could be to assign as much as possible, e.g:

```
REF (INT,INT) ::= REF 3;         /* REF (INT := 3, INT) */
{INT a; INT c} ::= {INT b; 3 c}; /* {INT a; INT c:=3} */
```

### B.4 Universal identifiers

A special identifier ? matching all names could have been provided. It can only be used in declarations, i.e. E ?. Its main use is in connection with dynamic search in environments, e.g. Smalltalk classes and exception handlers.

Note the similarity between declaration of a universal identifier, E ?, and making an exception continuation returning the same value, i.e.:

```
TRAP.undefid ::= BEGIN
  E univ;
  FUN -> sc//(/)
  -> reraise:cont:dval:env:status;
  cont//(univ)
  NUF(//)
END;
```
The difference is that \( ? \) hides through concatenation, i.e.

\[
\{E1\} \& \{E2?\}
\]

will hide all declarationss in \( E1 \). A restricted form of \( ? \) is provided through `catchall` used when searching for exceptions.

## B.5 Widening

Widening could be an operation to construct the least non-union constructable value out of a union of values. Some simple examples are:

\[
\begin{align*}
\text{WIDEN (3 UNION 4);} & \quad /* \text{INT} */ \\
\text{WIDEN (3 UNION TRUE);} & \quad /* \text{BOOL} */ \\
\text{WIDEN ({INT a; INT b} UNION {INT b; INT c});} & \quad /* \{\text{INT b}\} */
\end{align*}
\]

Widening is also well defined for aliases (and thus recursive values), and has the interesting effect of disassociating aliases which are not present in all components. Thus widening is kind of a dual to unification which makes more aliases. The examples of unification of recursive structures given section 3.8, also carries over to widening:

\[
\begin{align*}
\text{DIOV a := REF REF a;} \\
\text{DIOV b := REF REF REF b;} \\
\text{WIDEN (a UNION b) \quad /* DIOV x := REF REF REF REF REF x;}
\end{align*}
\]

Note that the "depth" of the unified value is the greatest common divisor of the "depth" of the argument for linear recursive values (\( \text{gcd}(3,2) = 1 \)). Widening gives the least common multiple (\( \text{lcm}(3,2) = 6 \)). The effect for non-linear is spectacular, the example in figure B.1 is taken from section 3.8.

The reason why widening was left out, was that it is not well defined for the static environment of function (and continuation) components. For copy and unification we were able to find a unique new denotable value for the denotable values which participated in the operations, but this has not been possible for widening. Apart from this the algorithm for widening is well defined.

## B.6 Non-determinism

Nel has no concept of non-deterministic choice, except for the implicit non-determinism introduced by the arbitrary interleaving of parallel processes. One form of non-determinism which seems interesting can be introduced in connection with set values, i.e. select one arbitrary component of the set for the operation. This is well defined for finite sets like \( \text{INT}, \text{CHAR} \) and \( \text{BOOL} \), but becomes more complicated...
for infinite sets like DIOV. What does it mean to call a DIOV function, follow a DIOV pointer, or open a DIOV environment? Whatever the answer might be, this form of non-determinism might have some connection to abstract interpretation, as it seems similar to one arbitrary instance of the collective interpretation.
Appendix C

Change oriented versioning

This appendix contains a short introduction to the main concepts of change oriented versioning. Change oriented versioning (COV) was introduced in [HOLAGER88], and has been applied to a software engineering data base in the EPOS project [LIE90]. The concepts in change oriented versioning are:

option; a boolean variable describing a functional change. Setting the option to true indicates that this functional change will be included.

choice; a setting of values for the options, describing a set of functional changes, i.e. a version.

change; a set of changes to the database to implement a combination of options.

visibility; a boolean expression over options. The database contains fragments with visibilities, and those visibilities which evaluates to true for a given choice are part of this version, i.e. visible.

ambition; a partial (incomplete, i.e. leaving some options unspecified) choice which is specified in connection with a change, and is used to change the visibilities of the changed fragments. It determines for which choices these changes will be visible. Operations with an ambition a on a database will give the following fragments and visibilities:

- insert: a new fragment with visibility \( v := a \)
- delete: only the visibility of the "deleted" fragment will be updated, i.e. \( v := v \land \neg a \).
- update: the visibility on the original fragment is updated to \( v := v \land \neg a \), and a new fragment which is a modification of the original gets visibility \( v := v \land a \).

Thus the ambition can be viewed as the scope or propagation of the changes.

validity; a boolean expression used to restrict the set of possible choices and ambitions.
The use of COV can be illustrated by the following small example. There are two options, \( o_1 \) and \( o_2 \). Let \( o_1 \) describe an optional feature and \( o_2 \) describe a bugfix. Then a new bug is reported, and a new option \( o_3 \) is introduced to denote this bugfix.

When the changes for \( o_2 \) are implemented it is natural to have an ambition \( o_2 \land o_3 \), i.e. the fixes should only be visible if all previous bugs are also fixed. This implies that we do not intend to make all bugfixes orthogonal. But \( o_1 \) is left unspecified in the ambition, which indicates that the changes will be visible regardless of whether we set \( o_1 \) to true or false. Thus the bug is not dependent on the inclusion or exclusion of the orthogonal feature.

The choice must be within the ambition, thus we can only choose the value of \( o_1 \). As the ambition determines the set of versions which are affected by the current changes, the choice determines which of these versions we see when we do the changes. If we for instance set \( o_1 \) to true, we will not see fragments changed or deleted to implement \( o_1 \), while implementing the changes. Thus we may need to set \( o_1 \) to false in the choice afterwards to make some additional changes. The choice can leave \( o_1 \) unspecified if it is possible to edit more versions simultaneously as in DEI (section 4.6)

The main advantages of COV are:

- The ease of combination of changes, i.e. merging of versions. Any combination of options gives a version.

- The additional changes necessary to merge two changes will only be visible when both changes are wanted, and need not get a new name.

- The orthogonality of versioning to the product structure, i.e. when a choice is specified one version of the total data base is selected.

- The specification of which versions are affected by the changes through the ambition.

- Selecting the version based on functional characteristics, and not on version number, i.e. a change spanning more modules are still characterized by the same option.

Note that COV is similar to the SCCS deltas and conditional compilation in ordinary preprocessors. The original aspect is the introduction of functional change as the primary concept, the use of ambitions to restrict the extension of changes, and the use of validities to restrict choices. COV is treated in more depth in Lie's thesis. In section 4.6 the concepts of change oriented versioning is used to version program text.

In [SARNAK88] concepts very similar to COV are introduced:

- The ambition is called the mask and the selected versions the edit set.

- The specific version within the edit set which is edited is called the观ew.
• The visibility is called the *conditional expression* and is attached to each fragment.

One difference is that in [SARNAK88] they use not only binary but multivalue variables.
Appendix D

Types

This appendix gives a short overview of some work on types in programming languages. It is mainly an overview, but it also contains some comparision with the solutions chosen in Nel.

Types are in general constraints on the values with which we compute. Values have certain types, defining which operations are allowed on them, and what result we can expect when performing these operations. Variables are also certain kind of values, where we can store other values, and their type specifies which values we are able to store there.

The reason why we type our values are several:

- It avoids misinterpreting the values, thus detecting errors in the code before or during run time.
- More efficient code can be generated, i.e. the compiler can statically determine if an operation is allowed, and the size of the values.
- The type can be considered as an invariant of the values, i.e. properties or restrictions, helping to understand the program.
- The type can be seen as a contract between the implementer and the user, thus helping in modularization and data abstraction.

The structure of this section is as follows. In section D.1 we discuss aspects of primitive and composite non functional types. The concepts of type construction, subtyping and abstract types are discussed. Section D.2 is devoted to functional types. The challenge here is to express as much information as possible, about an as flexible function as possible, in a compact notation which is decidable. We treat simple function types, general predicates, polymorphic functions and type parameters.
D.1 Non functional types

Non-functional types represent data objects, and not code. Even if the distinction is blurred in constructs like classes and modules we can usually distinguish between those values which can be executed and those which can’t.

Type construction

Non-functional types are usually constructed from basic types with type constructors. Different examples of type constructors are built in ones like arrays, records and unions, and the user defined ones in functional and logical languages for building recursive tree like data structures. The class concept in object oriented languages is an extension of the record concept in ordinary imperative languages.

Note that the type constructors in most languages are quite orthogonal, i.e. all constructed types have the same status as the built in ones, and can be used as components of new constructed types. E.g. a record can contain an array which elements are records which contain pointers to the main record.

Algol 68 contains constants, i.e. types with known value, and the type constructors can also be used to build composite constants.

Another important aspect of the constructed types is whether they have a predefined equality or assignment operation, and if they can be passed as arguments to functions and returned as results. As an example Pascal defines assignment for simple types and pointers, which can also be returned from functions, whereas arrays, records and function/procedures can in addition be passed as parameters.

Another question is how deep the operations of equality and assignment are. In Scheme there are for instance three different equality operations.

With most constructed types we can do all the checking during compile time, e.g. find type and offset of a record component. But some destructions require run time checks, e.g. checking the bounds before indexing an array and checking for nil when following a pointer. We also have to put out run time checks for partially defined operations, e.g. overflow on numbers.

Type equivalence

A type is intended to denote a set of values, and it is not obvious that a function written for one type should work for another, even if the types are similar, or that a variable defined to store values of one type could contain those of another. There are four kinds of type equivalence:

- Name, i.e. two different types can never be interchanged, this is used in Pascal. The advantage is that the language becomes type secure, but different types with the same representation need separately defined functions. An example:
TYPE Tab1 = ARRAY[1..10] OF Integer;
Range = 1..10;
Tab2 = ARRAY[Range] OF Integer;

Here Tab1 and Tab2 will be different.

- Structure, i.e. types are built up by the same constructor applied to com-
  ponents of equivalent structures can be interchanged. This achieves reuse
  of functions over equivalent types, but does not protect against unintended
  interchanging. Tab1 and Tab2 are equal.

- Behavior, i.e. all types which are operationally the same. This is the subtype
  notion used by Cardelli on records, and is similar to the equal predicate in
  Lisp. Note that it is undecidable for function values. Environments in Nel
  are behaviourally equivalent.

- Text, i.e. all types which are defined with the same text are equivalent. Here
  Tab1 and Tab2 will be different. This is similar to the type equivalence used
  in Russell, and it depends on the impossibility to redefine names and use
  variables in type constructions.

Subtyping

Subtyping is similar to type equivalence, as it defines a partial order on a set of
 types, whereas equivalence defines equivalence classes of types. Subtyping can be
 based on any of the type equivalences described in the previous paragraph.

The general idea of subtyping is to restrict the set (domain) of values in the
 subtype, i.e. all values which are of the subtype is also of the supertype. Subtyping
 is usually defined for three different types:

- Sets, i.e. subsets of integers, characters or user defined constants. Note that
  the storage needed for values of the subset is the same as for the original
  set, and the values of the subset can be used wherever the original values
  are used. But we must introduce run time checks in connection with storing
  in a variable which can only store a subset value, or we have to redefine
  the operations on elements of the original set for the subsets, so that they
  only return values in the subset range. The operations of the original set are
  usually used, and the check is performed before storing. This is similar to
  overloading.

- Functions. A function type can be viewed as a pre- and a post-condition
  which the function satisfies. Thus if the pre-condition is satisfied before the
  call, the post-condition will be satisfied after. The function can be refined
  by relaxing the pre-condition or strengthening the post-condition. The new
  function type can always be used with the input of the old, and the output
  from the new function type will always be within the output from the old.
• Records. A subrecord defines the same fields as the superrecord, but the type of some fields might be a subtype of the one in the original record. In addition the subrecord might define more fields. This is the subtyping used in object oriented languages. If the subrecord defines additional fields, it will need more space than the original record. Thus space for record variables can not be allocated statically, but has to be allocated on the heap.

Note that there is a general problem with subtyping and assignment. A subtype value can be used where ever a supertype value is expected, but any supertype value can not be stored in a subtype variable. Subtyping for record structures as defined by Cardelli extends the original Simula subtyping concept by allowing the attributes to be refined. This also creates an assignment problem. In Eiffel this problem is solved by making the exported variables from a class read only.

The laws defined on free data types in Miranda is also a form of subtyping, i.e. the laws makes equivalence classes of the freely generated terms.

Coercion and overloading

Some types are operationally similar but have a different machine implementation, e.g. integers and floating point numbers. Usually the operations with the same name are defined on these types, i.e. arithmetic operations. This is called overloading, or ad hoc polymorphism by Strachey. Overloading can also be defined explicitly in some languages, e.g. Ada. The compiler resolves the overloading by looking at the type of argument, and selecting the appropriate function. I.e. the matching of a function is not only on name, but also on type of parameter. A similar concept can be found in object oriented languages like Smalltalk, where the message is sent to the class of the object for interpretation. E.g. an expression like \(3 + 4\) will be interpreted as follows:

• An Ada compiler will during compile time discover that 3 and 4 are integers, and select the integer addition.

• A Smalltalk interpreter will see this as the (binary) message \(+ 4\) which is sent to the object 3, which is an instance of the class integer, where the appropriate method for integer addition will be found and applied.

In the definition of Haskell an attempt is made to formalize the concept of overloading, i.e. types belong to classes which define operations, which are "implemented" in each type. The classes belongs to a hierarchy.

When we mix different types for the same operation, e.g. adding a real and an integer, we need to coerce one of the arguments so that we can apply one of the operations. In the example with integers and reals compilers usually convert the integer to a real, and apply the real operation. This type of automatic coercion appeared in FORTRAN and was found very useful. In Algol 68 coercion between different values was used to a large extent, e.g. automatically following a pointer. This did not always give the result one expected. Few present languages have more
advanced forms of coercion than simple numerical. One exception is C++, where each class can define coercion functions, which are applied if the compiler finds that the argument has a wrong type.

Some languages also have the possibility to force a value to be interpreted as something else than the type it has. This is called casting. Casting takes advantage of the hidden representation of the value, and is highly unsafe.

**Abstract types**

Abstract types are types where only partial information of their representation is visible. Examples of abstract types are package in Ada, abstype in ML and Miranda, and class in Simula, Eiffel and Smalltalk. Abstracting away information about the representation enables the change of the implementation without affecting the user.

In Ada a type can be declared private. Private types only allow declaration, passing as parameters, assignment and equality test. Declaring a type as limited private also disallows assignment and equality test. This is generalized in Russell where the user has total control over the operations exported from any type.

Note that a variable declared with a private type has to be opened when it is passed to a trusted function, e.g. one inside the package where the private type is defined. This opening means that the type has two interfaces, and the compiler determines which one is appropriate. In Clu this conversion is explicit by up (from concrete to abstract) and down (from abstract to concrete). cvt is an abbreviation used for parameters and results. This conversion is automatic in Ada, where the type is abstract outside, and always open inside. The same conversion occurs in Smalltalk, i.e. when an object receives a message that object is opened by the class which has the corresponding method.

A refined form of abstraction is possible in C++ and Eiffel where the concepts of friend, private and protected are introduced. Friend indicates classes with additional access to some of the resources of a class. The difference between private and protected in C++ is that a protected resource is visible in subclasses, which is not the case for private resources.

All these constructs are syntactic abstractions enforced by the compiler, i.e. the explicit conversion between the external and internal interface. Syntactic abstractions are not easily to enforce in an interpreted environment. Thus any form of syntactic abstraction is left out in Nel.

Abstract types have been formalized by Girard [GIRARD72] and Reynolds [REYNOLDS74,85] as existential types, i.e. $\exists T. \Omega$, where $\Omega$ defines some operations on $T$. 

140
Abstraction and pattern matching

Functional and logical languages are heavily based on pattern matching, which requires that the internal representation of the types is visible. This defeats the idea of abstract types, and a good cooperation between these two concepts has not been found [WADLER87]. This is in a way similar to the difference between imperative implementations and algebraic specifications.

D.2 Function types

A function is an arbitrary many to one relation between its input and its output. The type is an abstract intentional description of this relation.

Simple function types

This is the traditional view of a function type which is found in most statically typed programming languages like e.g. Pascal. A function type is considered a mapping from one non-functional type to another, e.g. \texttt{fac(x):INT->INT}, or as written in Pascal \texttt{function fac(integer x):integer}. This typing tells us that given an integer the function \texttt{fac} will return an integer if it terminates. This can easily be expressed as a pair of pre and post conditions, i.e. \texttt{pre = parameter \in integer fac post = result \in integer}.

Higher order functions are also easily expressed in this form, e.g. \texttt{INT -> (INT -> INT)}. This scheme breaks down when we want to write functions which work on different kinds of arguments. Note that we have to specify the type of parameters and result completely, even if the function body only depends on part of the type structure, and could work equally well for many kinds of types. This is addressed in connection with polymorphic functions.

General predicates

More complete information of a function’s behaviour can be captured by general first order predicates, so called pre- and post-conditions. These could give the complete information about the semantics of the function, i.e. axiomatic semantics. The problem of checking these general conditions at compile time is undecidable. Some languages have included assert clauses (e.g. Eiffel), which are boolean functions which are invoked to check a condition at run time. This is similar to dynamic type checking, e.g. checking for array bounds. A clever compiler can also try to discover the result of these assertions during compile time, and thus either make the run time check redundant, or indicate a compile time error.

C also includes an assert macro facility, which compiles to a function. Anna [LUCKHAM84] is a separate assertion language developed for Ada. In the same group they have also developed a language, Task Sequencing Language (TSL)
[LUCKHAM87] for specification of sequencing constraints on concurrent processes.

Constructive logic

The idea in constructive logic is to look at the specification or type of the program as a theorem, and then the proof will be the program. To achieve the executability of the proof, the logic is restricted to what is called constructive logic, i.e. every proof contains an algorithm. A consequence of this restriction is that some ordinary logical rules like $p \lor \neg p \Rightarrow \text{true}$ are not valid, as they are not constructive. I.e. it does not give us a procedure on how to prove neither $p$ nor $\neg p$. To prove $p \lor \neg p$ we have to either prove $p$ or $\neg p$. Note that different proofs for the same theorem will correspond to different implementations of the same specification.

Polymorphic functions

As mentioned in connection with the typing of Pascal functions, the type of parameters had to be completely specified even if the body of the function only depended on some part of the type structure. Milner [MILNER78] used this observation when designing the type system for ML In ML every value and built in operation can be assigned an unique type. This information is used to deduce the types of variables and expressions. For functions the parameters are only restricted by the use in the function, and the functions type will be the most general.

Note that this implicit type inferencing depend on known basic types, and breaks down for modules where hiding is introduced.

ML's type inferencing precludes a generic function to be instantiated in two different ways in the same scope [HUDAK89], e.g;

```haskell
silly map f g
where f :: int -> int
      g :: char -> char
      map :: (a -> b) -> [a] -> [b]
    silly m f g = (m f num_list, m g char_list)
```

Attempts to extend the inferencing to allow the different instantiations are surveyed in [BARENDREGT90].

The most general type for an ML function can be seen as a universally quantified type over the type variables.

Polymorphic functions where the type of the result depend on the type of the argument, e.g. 'a list -> 'a list in ML, will also have subtypes. A subtype will be 'a -> 'a. I.e. any function returning a result of the same type as its argument will return a list with the same type. This seems a bit counterintuitive at first, as it seems like we are widening also the post condition, but since the post-condition depends on the pre condition, this is not the case.
The class concept in object oriented languages like Simula also introduces a restricted kind of polymorphic functions, i.e. when a function is written for one class, it will also work for all subclasses of this class.

**Explicit type parameters**

Languages like Ada have explicit type parameters to generic packages and subprograms. The syntax of Ada is designed so that the instantiation of generic packages is done during compile time.

Explicit type parameters are formalized in the second order typed lambda calculus with dependent types [GIRARD72], [REYNOLDS74], i.e. \( \prod x : A.B(x) \) with type \( \forall \tau \leq A.B(\tau) \) or \( \Delta \alpha . e^\alpha \) : \( \Delta \alpha . \omega \). These are powerful type systems, but the problem of determining if an expression is well typed is undecidable.

In [HUDAK89] the status of type checking for functions is summarized:

- Typed \( \lambda \)-calculus is strongly normalizing, i.e. its expressive power is restricted by not having general recursion or iteration.

- Adding constants \( Con \), and a fixed-point combinator constant for each type, i.e. \( \forall \tau Y_\tau : (\tau \rightarrow \tau) \rightarrow \tau \in Con \). This makes it the same as explicitly typed imperative languages like Pascal and Ada. E.g.

  \[
  fact = Y_{int}(\lambda f.\lambda n.n = 0 \rightarrow 1, f(n-1))
  \]

- Introducing

  type variables \( b \in BasTyp, v \in TypId, \tau \in Typ \)

  type formation rules \( \tau ::= b \mid v \mid \tau_1 \rightarrow \tau_2 \)

  extended beta-reduction rules \( (\lambda x^\tau . e_2^\tau) e_1^\tau \leftrightarrow (e_2^\tau[x^{\tau_1}/e_1^\tau])^{\tau_2} \)

  and \( (\lambda x^\tau . e_2^\tau) e_1^\tau \leftrightarrow (e_2^\tau[\tau_1/v][x^{\tau_1}/e_1^\tau])^{\tau_2} \)

  makes the type-checking for such a language undecidable, i.e. the type system is rich enough to express undecidable questions.

Even type inferencing in the ML system is NP-complete (POPL90), but it does not matter in practical cases.

A good overview of state of the art in typed \( \lambda \)-calculus is given in [BAREN-DREGT90].
Appendix E

Nel and Prolog

This appendix contains a rather complete translation of Prolog into Nel. It serves as an example of the use of Nel as an intermediate language.

Unification in Nel makes a new value which is the lub of the two arguments. It will always succeed, as VOID is the lub of all values. To check if the unification has succeeded in Prolog terms, we have to use VOIDED which checks if some component of a value is VOID. Unification in Nel preserves aliases in its arguments, but aliases from outside the arguments are lost, i.e. still refer to the old values. A simple example of this is given in section 2.3.5. We will represent the substitution of variables as an environment.

The following constructs are used to model Prolog in Nel:

- Ordinary Prolog terms are declared as:

```
DIOV name       /* variables */
name1 alias     /* aliases */
E name          /* constants, functions, sets etc. */
REF ( 'f', DIOV Y, DIOV X ) /* f(X,Y) in Prolog */
```

The extended terms defined in LOGIN [AIT-KACI86] are modelled as environments, and the unification in Nel will work automatically. The example from [AIT-KACI86] will be:

```
{child()} label;
{person()} label;
  queen() knows;
  monarch() hates} knows;
{child()} label;
  knows.hates knows;
  wickedqueen() likes} hates;
  knows like}

{adult() label;
```
\{adult() label;
  witch() knows\} knows;
\{person label;
  monarch knows;
  knows like\} hates\}

FUN child() UNION adult() NUF person;
FUN child() ~ adult() NUF teenager;

The unification will also work for all kind of other values in the Nel value
lattice, even for pointer structures. Thus we get a typed Prolog for free.

- The substitution is a nested environment with the bindings of all variables.
  This is passed along to each goal, and a new environment with the new bind-
  ings is returned if the goal was resolved successfully. The old environment
  is kept in the goal for backtracking.

- A clause is evaluated as follows:

\[ h(X,f(X,Y),Z) :- p(X,g(Y,c),V), q(V,W), !, s(b,W,Z) \]

- \( h \) is called with the parameters, the current substitution and the cur-
  rent continuation. It returns either (VOID, VOID) (failed), or a new
  substitution, and a continuation to find more solutions.

- The parameters and the current substitution are unified against the
  head, i.e.

  \[
  \text{param'}(\text{DIOV } X, \text{REF}'f', X, \text{DIOV}), \text{DIOV}, \text{DIOV}) \ X:p2:Z: \text{subst unif};
  \]

  Note the extra parameter which is all the old substitutions. The new
  substitution \text{subst} will now contain all the bindings.

- The substitution is checked, if it contains a VOID component the clause
  fails, and the next \( h \) is called with the same parameters, if this is the
  last \( h \) clause, the continuation is called with (VOID, VOID). If it suc-
  ceed a new substitution containing all the local variables and the old
  substitution is made, i.e.

\[ \{X X; (\text{CONT } p2)[3] Y; Z Z; \text{DIOV } V; \text{DIOV } W; \text{subst } \text{Old}\} \hs; \]

Furthermore a function calling the continuation if \( h \) should later fail is
made, i.e.

\[
\text{DIOV } \text{hc} := \text{FUN } h(\text{param})(\text{cont}) \text{ NUF}
\]

hc can later be changed if a cut is encountered.

- \( \hs \) is passed along with the right components to the first clause, i.e.

\[
\text{DIOV}, \text{DIOV } ps:pc:=\text{db}.p(\text{hs}.X, \text{REF}'g', \text{hs}.Y,'c'), \text{hs}.V, \text{hs})(//);
\]

\( p \) returns a new substitution and a continuation (to find the next sub-
  stitution). Note that the old substitution (\( \hs \)) is not changed, and that
  \( p \) is looked up in the \text{db} environment, thus starting the search from
  the start of the environment. Note also that if this definition of the clause
  \( h \) fails, the next one will be invoked when \( h \) is called by \( \text{hc} \).
- The subgoals are called in a loop to find all solutions, i.e.

```scheme
WHILE pc <> VOID DO
    /* resolve other subgoals */
    ps:pc := pc//()// /* try next */
    OD;
```

- The cut, !, changes the continuation of the head and all the previously resolved subgoals, i.e.

```scheme
hc := FUN cont(VOID,VOID) NUF;
pc := FUN ->//()//(VOID, VOID) NUF//();
    /* simulating a fail */
```

The making of a failing continuation is quite complex. The function uses the parameter as a continuation, calling it with its continuation. Thus the function returns a continuation (//(VOID,VOID)) which uses the parameter as a continuation and returns (VOID,VOID) to it. If we had intermixed the syntax for invoking continuations and functions (as in Scheme) we could have used FUN VOID, VOID NUF. Note though that the cut continuation is the same for all, so that we can make it once, and use it as a constant cutcont.

- When the last subgoal is successfully resolved, the continuation, cont, of h is called with the current continuation and the current binding, i.e.

```scheme
cont := cont//(ss.Old)
```

The complete code should now be:

```scheme
FUN -> param;
FUN =: DIOV cont;
param := (DIOV X,REF('f',X,DIOV),DIOV,DIOV) X:p2:Z:subst unif;
IF NOT VOIDED unif THEN
    {X X; (CONT p2)[3] Y; Z Z; DIOV V; DIOV W; subst Old} hs;
    DIOV hc := FUN h(param)(cont) NUF;
    DIOV,DIOV pc:ps:=db.p(hs.X,REF('g',hs.Y,'c'),hs.V,hs)//();
    WHILE pc <> VOID DO
        DIOV,DIOV qc:qs := db.q(ps.V, ps.W, ps)//();
        WHILE qc <> VOID DO
            hc := FUN cont//(VOID,VOID) NUF;
            pc := cutcont;
            qc := cutcont;
            DIOV,DIOV sc:ss := db.s('b', qs.W, qs.Z, qs)//();
            WHILE sc <> VOID DO
                cont := cont//(ss.Old);
                sc,ss := sc,//()
            OD;
            qc,qs := qc//()
            OD;
        pc, qc := pc//()
    OD;
```
ELSE
    h(param)(cont) /* cont;//(VOID,VOID) if last clause */
    FI
    NUF
    NUF h:

- Assert and retract are not defined in this emulation. They change the db environment. A modified form of assert is to extend the environment with a new definition at the end (which will mean that the asserted definition is found first, as the environment is searched from the end backwards), and to retract all clauses by UNDEF, i.e.

    db.g oldg;
    db := db & {FUN ... oldg ... NUF g};
    db := bd & {UNDEF g};

Note how the new clause is linked onto the old one by declaring an alias before db is changed. We cannot assert a goal at the begin of the environment as it will not be called by the first goal, i.e. the bindings used for backtracking are the static ones in the environment.

- In Prolog a term can be used as a goal, i.e. data is interpreted as code. An example:

    f(a,b).
    g(X) :- X.
    ?- g(f(X,Y))

This is easily emulated in Nel by letting the constants be functions, i.e. REF(FUN ->.f NUF, DIVV, DIVV) for f(X,Y), and when a variable appear as a goal it is used as (CONT X) hd:tl; hd(db)(tl,subst)//). In LOGIN the functor name can be a set, i.e. adult, and it is not clear how it is interpreted as a goal. In Nel a union value cannot be called. Probably it is counterintuitive to interpret it as a goal because the type structure actually represents the following information:

    person(X) :- child(X).
    person(X) :- adult(X).
    teenager(X) :- adult(X), child(X).

The inclusion of types in an untyped language, will certainly make the interchange of code and data more difficult.
Bibliography
Bibliography

[ABRAMSKY87] Abstract interpretation of declarative languages
Samson Abramsky, Chris Hankin (Eds)
Ellis Horwood Limited, 1987

[AIT–KACI86] Logic and inheritance.
Hassan Aït–Kaci, Roger Nasr
In Proceedings 13'th ACM POPL, pp.219–228, 1986

[AHO86] Compilers – Principles, Techniques and Tools
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Addison-Wesley, 1986

[ALBANO85] Galileo: A Strongly-Typed, Interactive Conceptual Language
Antonio Albanese, Luca Cardelli, Renzo Orsini
ACM Transactions on Database Systems, Vol. 10, No. 2, June 1985

Pierre America, Jaco de Bakker, Joost N. Kok, Jan Rutten

American National Standards Institute, Inc. ANSI/MIL–STD–
1815A–1983

[BARENDREGT90] Types in Lambda Calculi and Programming Languages
Henk Barendregt, Kees Hemerik
3rd European Symposium on Programming - ESOP’90, , N.
Jones (Ed), LNCS 432, pp. 1-35

[BRATSBERG88] The Nel Interpreter Implementation
Svein Erik Bratsberg, Erik Odberg
Norsk Informatikk Konferanse, NIK 88, pp. 171-185, Tapir 1988

[CARDELLI84] A semantics for multiple inheritance.
Luca Cardelli
in LNCS 173, G. Kahn et.al. (Eds), pp.51–67, Springer 1984
[CARDELLI89] Operations on Records
Luca Cardelli, John C. Mitchell
Category Theory and Computer Science, LNCS 389, pp. 75-81, Springer 1989

[CHOMSKY56] Three models for the description of language
N. Chomsky
IRE Transactions on Information Theory, IT-2:3, pp.113-124

[COLMERAUER70] Les systèmes q’ou un formalisme pour analyser et synthétiser
des phrases sur ordinateur
Alain Colmerauer
Intern. Rep. 43, Département d’informatique, Université de Montreal, Canada, Sept. 1970

[COLMERAUER73] Un système de communication homme-machine en français
Alain Colmerauer, H. Kanoui, P. Pasero, P. Poussel
de Luminy, Marseilles, France, 1973

[COLMERAUER82] Prolog and infinite trees
Alain Colmerauer
231-251, Academic Press, 1982

[COURINGTON89] The Network Software Environment
William Courington
Sun Technical Report, 104 p, 1989

[DAHL70] Common Base Language
O. J. Dahl, B. Myhrhaug, Kristen Nygaard

Alan Demers, James Donahue
In Proceedings 10’th ACM POPL, pp.59–72, 1983

[DEREMER76] Programming-in-the-Large versus Programming-in-the-Small
F. DeRemer, H. Kron
IEEE Transactions on Software Engineering, June 1976, pp. 321-
327

[DEUTSCH90] On determining lifetime and aliasing of dynamically allocated
data in higher-order functional specifications
Alain Deutsch

[DIJKSTRA68] Co-operating sequential processes.
E.W.Dijkstra
43–112, 1968
[FELDMAN79] Make – a program for maintaining computer programs
S. I. Feldman
Software – Practice and Experience, )((3), pp. 255-265, March
1979

[FILINSKI89] Declarative Continuations: an Investigation of Duality in Pro-
gramming Language Semantics
Andrzej Filinski
Category Theory and Computer Science, pp. 224-249, LNCS 389,
Springer

[FLOYD67] Assigning Meaning to Programs
R. W. Floyd
Proceedings of Symposia in Applied Mathematics 19, pp. 19-32,
1967

[GABRIEL88] Draft Report on Requirements for a Common Prototyping Sys-
tem.
Richard P. Gabriel (Ed)
SIGPLAN Notices, Vol. 24, No. 3

[GIRARD72] Interprétation Functionelle et Elimination des Coupures dans
l'Arithmétique d'Ordre Supérieur
J.-Y. Girard
Thèse de Doctorat d'Etat, Paris, 1972

[GOLDBERG83] Smalltalk-80: The language and its Implementation
Goldberg A., Robson D.
Addison-Wesley, 1983

[GOOS83] DIANA an intermediate language for Ada.
G. Goos, W.A. Wulf, A. Evans Jr., K.J. Butler
LNCS no. 161, Springer–Verlag, 1983

[GULLA90] Change Oriented Version Descriptions in EPOS
Björn Gulla, Even-André Karlsson, Dashing Yeh

R.H.Halstead Jr.

[HEERING85] Towards monolingual programming environments
Jan Heering, Paul Klint
TOPLAS, 7, 2, April 1985, pp. 183-213

[HOARE69] An axiomatic basis for computer programming.
C. A. R. Hoare
[HOLAGER88] Elements of the Design of a Change Oriented Configuration Management Tool  
Per Holager  
Technical Report STF44 A88023, ELAB, NTH

[HUBER87] EPOS – Extended Prolog with Order-Sorted Resolution  
Martin Huber, Igor Varsek  
Interne Bericht 4/87, Institut für Informatik I, Universität Karlsruhe

[HUDEK88] Report on the Functional Programming Language Haskell  
P. Hudak, P. Wadler (Eds)  
Tech. rep. YALEU/DCS/RR656, DCS, Yale University

[HUDEK88] Conception, Evolution, and Application of Functional Programming Languages  
Paul Hudak  

[JENSEN74] Pascal User Manual and Report  
K. Jensen, N. Wirth Springer Verlag, 1974

[JONES77] The narrowing gap between language systems and operating systems  
Anita K. Jones  

[KARLSSON88] Types and Environments as first class objects  
Even-André Karlsson  
Norsk Informatikk Konferanse, NIK 88, pp. 157-169, Tapir 1988

[KENAH84] VAX/VMS Internals and Data Structures  
Lawrence J. Kenah, Simon F. Bate  
Digital Press, 1984

[KNUTH68] Semantics of Context-Free Languages  
Donald E. Knuth  
Mathematical System Theory, 2, 1968, pp. 127-145

[KRISTENSEN83] Abstraction mechanisms in Beta programming language.  
Bent Bruun Kristensen, Ole Lehmann Madsen, Birger Møller-Pedersen, Kristen Nygaard  

[LAMPSON83] Practical use of a polymorphic applicative language.  
Butler W. Lapson, Eric E. Schmidt  
[LANDIN65] A correspondence between ALGOL 60 and Church’s lambda notation
P. J. Landin
Communication of ACM, 8, 89-101. 158-165

[LANDIN65] The next 700 programming languages
P. J. Landin
Communication of ACM, 9, 3, pp. 157-165

[LIE90] Versioning in Software Engineering Databases
Anund Lie

[LISKOV77] Abstraction mechanisms in CLU
B. Liskov, A. Snyder, R. Atkinson, C. Schaffert
CACM, Vol.20, No.8, pp.564–576, Aug. 1977

[LISKOV82] Guardians and actions: Linguistic support for robust, distributed programs.
B. Liskov, R. Scheifler
ACM TOPLAS, Vol.5, No.3, July 1983, pp.381-404,

[LUCKHAM84] An Overview of Anna – A Specification Language for Ada
David Luckham, Friedrich W. von Henke
Tech. Rep. 84-265, Stanford, 1984, 22 p

[LUCKHAM87] Task Sequencing Language for Specifying Distributed Ada Systems – TSL-1
David Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, M. A. Haberler

[MCCARTHY60] Recursive functions of symbolic expressions and their computation by machine, Part I
John McCarthy
Communications of ACM 3, 4, 184-195

[MCCARTHY63] A basis for a mathematical theory of computation.
in Computer Programming and Formal Systems, North Holland, pp. 37-70

[MEYER88] Eiffel: A Language and Environment for Software Engineering.
Bertrand Meyer

[MILNER78] A Theory of Type Polymorphism in Programming
Robin Milner
[MILNER84] A proposal for standard ML.
Robin Milner
1984 ACM Symposium on Lisp and functional programming, pp.184–197

[NAUR63] Revised report on the algorithmic language Algol 60
P. Naur (Ed)
Communications of the ACM, 6:1, pp. 1-17

[PCTE86] PCTE: A Basis for A Portable Tool Environment
PCTE Project Report
ESPIRIT '86: Results and Achievements, Elsevier, 1986

[PRIETO-DIAZ86] Module Interconnection Languages
Ruben Prieto-Diaz, James M. Neighbors

[REYNOLDS74] Towards a Theory of Type Structures
John C. Reynolds

[REYNOLDS85] Three approaches to type structure
John C. Reynolds
TAPSOFT 85, Berlin pp. 97-138, Vol 1, LNCS 185, Springer-Verlag

[RITCHIE74] The UNIX Time–Sharing System
Dennis M. Ritchie, Ken Thompson

[ROCHKIND75] The Source Control System
M. J. Rochkind

[SARNAK88] Creation and maintenance of Multiple Versions
N. Sarnak, R. Bernstein, V. Kruskal

[SCOTT70] Outline of a Mathematical Theory of Computation
Dana S. Scott
Technical Monograph PRG-2, University of Oxford, 1970

[SCOTT76] Data types as lattices.
Dana S. Scott
[SHAPIRO87] Concurrent Prolog: Collected Papers, Volume 1 and 2
Ehud Shapiro (Ed)

[SHIVERS88] Control Flow Analysis in Scheme
Olin Shivers
ACM SIGPLAN’88 Conference on Programming Language Design and Implementation, pp. 164-174, 1988

[SHIVERS90] Data-Flow Analysis and Type Recovery in Scheme
Olin Shivers

[SLOMAN83] Poplog: A Multilanguage Program Development Environment
A. Sloman, S. Hardy, J. Ginson

[STALLMAN81] EMACS: The Extensible Customizable, Self-Documenting Display Editor
Richard M. Stallman
ACM SIGPLAN SIGOA Symposium on Text Manipulation, June 1981

[STONEMAN80] Requirements for the Programming Environment for the Common High Order Language, STONEMAN

[STOY77] Denotational semantics: The Scott–Strathey approach to programming languauges theory.
J.E.Stoy
MIT Press, pp.414, 1977

[STRACHEY66] Towards a Formal Semantics
C. Strachey
in Formal Language Description Languages, Steel (Ed), North-Holland, 1966

[STRONG58] The problem of programming communication with changing machines: a proposed solution

Bjarne Stroustrup

[TEITELMAN81] The Interlisp Programming Environment
Warren Teitelman, Larry Masinter
IEEE Computer, 14:4, April 1981
[TEITELMAN84] A tour through Cedar
   Warren Teitelman
   IEEE Software, April 1984

[TICHY82] Design, implementation, and evaluation of a revision control sys-
   tem.
   Walter F. Tichy
   Proceedings of the 6th International Conference on Software En-
   gineering, IEEE, September 1982

[TURNER86] An Overview of Miranda
   David Turner
   Sigplan Notices, December 1986

[WADLER87] Views: A way for pattern matching to cohabit with data ab-
   straction
   Philip Wadler
   14th ACM Symposium on Principles of Programming Lan-
   guages, pp. 307-313, 1987

[WIJNGAARDEN75] Revised report on the algorithmic language Algol 68.
   A. van Wijngaarden et. al.