Design and Implementation of a Partial Evaluation-Based Compiler for an Asynchronous Realtime Programming Language

Markus Freericks

May 25, 1996

Abstract

This thesis describes a compiler for the asynchronous real-time programming language ALDiSP. Though the language has a complex semantics not suited for easy compilation, the compiler has to generate code for target platforms that have stringent space limitations, and for target applications that have to satisfy hard real-time requirements. To accomplish this feat, the compiler is based upon an *abstract interpreter* that simulates all possible evaluation paths of the program. In a reconstruction phase, the program is then totally re-created from the information gained during this simulation. The abstract interpreter is an extension of the formal semantics of ALDiSP.

Diese Dissertation beschreibt einen Compiler für die asynchrone Echtzeit-Programmiersprache ALDiSP. Obwohl diese Sprache eine komplexe Semantik hat, die eine direkte Übersetzung ausschließt, soll der Compiler Code für Zielarchitekturen erzeugen, die starken Speicherplatz-Restriktionen unterliegen, und für Applikationen, die harte Echtzeitanforderungen erfüllen müssen. Um dieses Ziel zu erreichen, enthält der Compiler als zentrale Komponente einen *abstrakten Interpreter*, der sämtliche möglichen Ausführungspfade eines Quellprogramms analysiert. In einer Rekonstruktionsphase wird das Programm aus den vom Interpreter erzeugten Annotationen vollständig neu aufbaut. Der abstrakte Interpreter ist eine Erweiterung der formalen Semantik von ALDiSP.

Contents

0 Introduction **7**

0.1 Real-Time Digital Signal Processing . . . 7
0.2 Application Areas for ALDiSP . . . 8
0.3 Overview of Compiler and Thesis . . . 9
0.4 Goals and Results . . . 10
0.5 Related Work – Languages . . . 11
0.6 Related Work – Partial Evaluation . . . 11
0.7 Related Work – Partial Evaluation for Numerics and DSP . . . 11
0.8 Stylistic Questions, Thanks, &c. . . . 12

1 ALDiSP – A Primer **13**

1.1 Lexical Structure . . . 13
1.2 Overall Structure . . . 13
1.3 Evaluation Strategy . . . 14
1.4 Basic Types . . . 14
1.5 Auto-Mapping . . . 14
1.6 Suspensions . . . 15
1.7 Blocking Access . . . 16
1.8 Sequences . . . 17
1.9 Multiple Return Values . . . 17
1.10 Conditionals . . . 17
1.11 Local Definitions . . . 18
1.12 Checking and Casting Types . . . 18
1.13 Exceptions . . . 18
1.14 Lists, Streams, Pipes . . . 19
1.15 Declarations . . . 20
1.16 Modules . . . 21
1.17 Functions . . . 21

2 The Parser 22

2.1 Overview 23

2.2 Lexical Analysis 23

2.3 Preprocessing 24

2.4 The Parser 25

2.5 Blind Alleys 27

2.5.1 Representation of Literals 27

2.5.2 Implementing The Preprocessor 28

2.5.3 Error Handling 29

2.5.4 Line Numbers and File Names 29

2.5.5 CPS Intermediate Form 30

2.5.6 CPS Transformation: The Morale 31

3 The Lambda Form 32

3.1 Semantic Domains 33

3.1.1 Conventions and Notation 34

3.1.2 Data Domains 35

3.1.3 Results 37

3.1.4 Evaluation Functions: an Overview 37

3.1.4.1 $\texttt{eval}_{\text{expr}}$ 38

3.1.4.2 `apply` 38

3.1.4.3 $\texttt{eval}_{\text{decl}}$ 38

3.1.4.4 $\texttt{eval}_{\text{program}}$ 38

3.1.5 State 39

3.2 Auxiliary Semantic Functions 40

3.2.1 $\texttt{eval}_{\text{exprs}}$ 40

3.2.2 $\texttt{deref}_{\text{objs}}$ 40

3.2.3 `block` 41

3.2.4 `strict`, $\texttt{strict}_{\text{exprs}}$ 41

3.2.5 $\texttt{eval}_{\text{thunk}}$, $\texttt{norm}_{\text{result}}$ 42

3.3 Evaluating Expressions 42

3.3.1 `Lambda`*: Closure Creation* 42

3.3.2 `Lvar`*: Variable Look-Up* 42

3.3.3 `Lit`*: Literal Values* 43

3.3.4 `Lapp`*: Function Application* 43

3.3.5 `Lcheck`*: Asserting Types* 44

3.3.6 `Lcast`*: Casting Types* 45

3.3.7 `Lcond`*: Two-Way Conditional* 46

3.3.8 `Lselect`*: N-Way Conditional* 46

3.3.9 `Lseq`*: Sequences of Expressions* 47

3.3.10 `Lcatch`*: Handling Exceptions* 47

3.3.11 `Let`*: Local Definitions* 47

3.4 Evaluating Declarations 48

3.4.1 `Ldecl`*: The Simple Declarations* 48

3.4.2 `Lpard`*: Parallel Declarations* 48

3.4.3 `Lseqd`*: Sequential Declarations* 48

3.4.4 `Lfixd`*: Recursive Declarations* 49

3.5 Application Rules 49

3.5.1 $\texttt{apply}_{\text{closure}}$ 50

3.5.2 $\texttt{apply}_{\text{overloaded}}$ 51

3.5.3 $\texttt{apply}_{\text{primitive}}$ 52

3.5.4 $\texttt{apply}_{\text{array}}$ 53

3.5.5 $\texttt{apply}_{\text{type}}$ 54

3.6 Auto-Mapping 54

3.6.1 $\texttt{apply}_{\text{mappable}}$ 55

3.7 Top-Level Evaluation 57

3.7.1 $\texttt{eval}_{\text{program}}$ 57

3.7.2 `schedule` 57

3.8 Discussion of Selected Problems 59

3.8.1 Macros 59

3.8.2 Auto-Mapping 60

3.8.3 Blocking 60

3.8.4 Implementing Recursion 61

4 Transformation of the AST into Lambda Form **62**

4.1 Transforming Programs: $\mathtt{T}_{\text{program}}$ . . . 63

4.2 Transforming Declarations: $\mathtt{T}_{\text{decl}}$ . . . 63

4.2.1 `Asimpledecl` . . . 63

4.2.2 `Aparamdecl` . . . 63

4.2.3 `Amultidecl` . . . 65

4.2.4 `Amoduledecl` . . . 66

4.2.5 `Aimportdecl` . . . 66

4.2.6 `Aabstypedecl` . . . 67

4.2.6.1 *Parameters* . . . 67

4.2.6.2 *An Example Transformation* . . . 68

4.2.6.3 *The Rules* . . . 69

4.2.7 `Arecdecl` . . . 69

4.3 Transforming Type Expressions: $\mathtt{T}_{\text{type}}$ . . . 70

4.3.1 `Afunctype` . . . 70

4.3.2 `Aexprtype` . . . 70

4.4 Transforming Expressions: $\mathtt{T}_{\text{expr}}$ . . . 70

4.4.1 `Avar` . . . 70

4.4.2 `Aapp`, `Acond`, `Acheck` . . . 70

4.4.3 `Acast` . . . 71

4.4.4 `Adelay` . . . 71

4.4.5 `Asuspend` . . . 71

4.4.6 `Atuple` . . . 71

4.4.7 `Aseq` . . . 71

4.4.8 `Astring`, `Aint`, `Afloat`, `Aconst` . . . 72

4.4.9 `Aguard` . . . 72

4.4.10 `Areturn` . . . 73

4.4.11 `Alocal` . . . 73

4.5 Post-processing Passes . . . 73

4.5.1 *Implementation of post-processing steps* . . . 73

4.5.2 *Expanding Macros* . . . 73

4.5.3 *Declaration Simplification* . . . 74

4.5.4 *Removing unnecessary Checks and Casts* . . . 76

4.5.5 *α-Conversion* . . . 76

4.5.6 *Problems with Rearranging Expressions* . . . 76

4.5.7 *Lambda Hoisting* . . . 76

4.5.8 *Transforming Free Variables into Parameters* . . . 77

4.5.9 *Strictness Analysis* . . . 78

5 Partial Evaluation – Definitions and Known Results **79**

5.1 Definitions . . . 79

5.2 Partial Evaluation, Residual Program . . . 79

5.3 Self-Application and the Futamura Projections . . . 80

5.4 Internal Specialization . . . 81

5.5 On-line/Off-line Partial Evaluation . . . 81

5.6 Static and Dynamic Binding Times . . . 81

5.7 Partially Static Values . . . 82

5.8 Specialization . . . 82

5.9 Generalization . . . 82

5.10 Folding/Unfolding . . . 82

5.11 Fixed Points . . . 83

5.12 Applying the Techniques of Abstract Interpretation to ALDiSP . . . 83

6 Abstract Interpretation of LF Expressions 85

6.1 Goals of Abstract Interpretation 85

6.2 Abstract Values 86

6.2.1 General Remarks 86

6.2.2 Domains 87

6.2.3 Basic Domain Constructions 87

6.2.4 Basic Examples 88

6.2.5 Union of Domains 89

6.2.6 The Ideal Abstract Domain and its Subsets 89

6.2.7 Size of Domains 91

6.2.8 Sources of Abstract Values 92

6.3 An Example 93

6.3.1 Abstract Conditionals 93

6.3.2 Loop Detection – Finding Fixed Points 94

6.3.3 Speed 95

6.3.4 Generalizing the Argument Lists 95

6.3.5 Conclusion of the Example 96

6.4 Implementing the Loop Detector 96

6.4.1 Naïve Loop Detection Schemes 97

6.4.2 Problems with the Naïve Scheme 98

6.4.2.1 Not Enough Generalization 98

6.4.2.2 Too much Generalization 98

6.4.2.3 Mutual Recursion 99

6.4.2.4 Multivariant Specialization 100

6.4.2.5 Restarting Approximations 100

6.4.3 Generalizing Loop Detection 101

6.4.4 Concrete Arguments 103

6.4.5 Evaluation-Order Effects 104

6.5 ALDiSP-specific Problems 104

6.5.1 State and Side Effects 104

6.5.2 Promises 105

6.5.3 Recursive Definitions 105

6.5.4 Exceptions 106

6.5.5 Function Values 106

6.5.6 Generalizing Numeric Types 108

6.5.7 Raising and Catching Exceptions 108

7 The Abstract Scheduler 110

7.1 Suspensions and other Schedulable Entities 110

7.2 Non-Determinism 112

7.3 Sequentializing the Schedule 112

7.4 Garbage Collection 114

7.5 Finding Loops 115

7.6 Alternative States 116

7.7 Blocking 116

8 Execution Trace Reconstruction **120**

8.1 The Code Form . . . 121
8.1.1 Differences between CF and LF . . . 121
8.1.2 Abstract Syntax of CF . . . 122
8.1.3 Notation . . . 123
8.1.4 Closures, Tuples, Combinators . . . 124
8.1.5 Top-Level Bindings . . . 124
8.1.6 Semantics of the Code Form . . . 124
8.1.6.1 Programs . . . 125
8.1.6.2 Atoms . . . 126
8.1.6.3 Primitives . . . 126
8.1.6.4 Conditionals . . . 127
8.1.6.5 Function Application . . . 127
8.1.6.6 Throwing and Catching Exceptions . . . 127
8.1.6.7 Evaluating Declarations . . . 128
8.1.6.8 Evaluating `Code` *expressions* . . . 128

8.2 Relation between Abstract Results and Code Attributes . . . 128
8.2.1 The Treatment of State . . . 128
8.2.2 `atom`*: a Code Attribute for Objs* . . . 129

8.3 Example: Straight-Line Code . . . 129

8.4 Inter-Function Optimizations . . . 133
8.4.1 Inlining . . . 134
8.4.2 Grouping . . . 135
8.4.3 Tabulation and Caching . . . 135
8.4.4 Elimination of Parameters and Return Values . . . 136
8.4.5 Variable Splitting . . . 136

8.5 Basic Optimizations . . . 137
8.5.1 Variable Propagation . . . 137
8.5.2 Dead-Code Removal . . . 137
8.5.3 Reference Tracing . . . 138
8.5.4 Tuple Tracing . . . 138

8.6 Example: Blocking Code . . . 139
8.6.1 What's in a `Block` . . . 139
8.6.1.1 Alternatives to Blocks: CPS . . . 140
8.6.1.2 Alternatives to Blocks: Syntax Transformations . . . 141
8.6.2 The Blocking Example . . . 142

8.7 Specification of Code Generation . . . 145
8.7.1 $\texttt{eval}_{\text{exprs}}$ . . . 146
8.7.2 $\texttt{deref}_{\text{objs}}$ . . . 146
8.7.3 `block` . . . 146
8.7.4 `strict` *and* $\texttt{strict}_{\text{exprs}}$ . . . 146
8.7.5 $\texttt{eval}_{\text{thunk}}$ *and* $\texttt{norm}_{\text{result}}$ . . . 146
8.7.6 Lit . . . 146
8.7.7 Lexical Lvars . . . 147
8.7.8 Dynamic Lvars . . . 147
8.7.9 Lambda . . . 147
8.7.10 Lcheck . . . 147
8.7.11 Lcast . . . 148
8.7.12 Lcond . . . 148
8.7.13 Lselect . . . 148
8.7.14 Lseq . . . 148
8.7.15 `apply` . . . 148
8.7.16 $\texttt{apply}_{\text{type}}$ . . . 149
8.7.17 $\texttt{apply}_{\text{primitive}}$ . . . 149
8.7.18 $\texttt{apply}_{\text{closure}}$ . . . 150

8.8 Related Work . . . 150

9 The Back-End **153**

9.1 Differences between M and CF 153

9.1.1 Disjoint Address Space, Index Types, and Hardware Loops 154

9.1.2 Data Formats, Special Instructions, and the ALDiSP *Library* 154

9.1.3 No Implicit State 155

9.2 Semantic Entities of M 155

9.2.1 Function Definitions 156

9.3 Structure of M 157

9.4 Transforming CF into M 159

9.5 The C back-end 160

9.6 The Future: Automatic Back-End Generation 160

10 Conclusions and Outlook **161**

10.1 Development History 162

10.2 Design Alternatives 162

10.2.1 Explicit State Passing 162

10.2.2 Continuation-Passing Style 163

10.2.3 Compilation by Graph Reduction 163

10.3 Redesigning ALDiSP 164

A Literature **165**

B Index **172**

Chapter 0

Introduction

Whenever anyone says, "theoretically", they really mean, "not really".
— Dave Parnas

This work describes the overall design and the internal structure of `ac`, the first compiler for the ALDiSP programming language. ALDiSP was developed by the author and Alois Knoll in 1989-90 at TU Berlin within the context of the CADiSP [73] project. The goal was to create an interactive, visually oriented system for the development of digital signal processing (DSP) algorithms based on a dataflow paradigm. In such a system, the user describes the algorithms by connecting parameterizable 'black boxes'. These basic components were to be described in ALDiSP (which, in turn, should be compiled into the imperative language ImDiSP [76] which now has a life – and a compiler – of its own).

It soon became clear, however, that the issues concerning the compilation of both applicative and imperative languages into high quality code for real-time applications were much more interesting than the original project goal. This also applies to the adaptation of those languages both to the requirements of real-time processing and efficient compilation. For this reason, the whole project was retargetted to the development of ALDiSP and ImDiSP.

0.1 Real-Time Digital Signal Processing

Digital signal processing (DSP) is concerned with the generation, transformation and analysis of (physically interpreted) signals, e.g. in audio applications. Typical DSP algorithms are

- *filters with constant coefficients* with finite or infinite response behaviour (FIR and IIR),
- *adaptive filters* used to cope with time-varying input signal characteristics,
- *signal transforms*, especially the family of *fast Fourier transformations* (FFT) algorithms, which are used to convert signals into the frequency/phase domain and back.
- *correlation functions* with tabulated sample signals, and
- *detection and estimation* of statistical signals/spectra.

These algorithms are usually defined as functions of equitemporally sampled signals. Their overall structure tends to be synchronous and quite simple. Algorithms working with asynchronous signals or with signals of different or varying sample rates are usually a lot more complex to describe, implement and analyze.

Another important application field of DSP techniques is image processing (computer vision, CV), i.e. the analysis and transformation of video signals. CV algorithms usually need so much computing power and memory bandwidth that real-time applications must be implemented using dedicated hardware.

ALDiSP was designed as a language especially suited for the functional specification of DSP applications.

ALDiSP is described in full detail in [41] and (less detailed) in [42]. Chapter 1 will explain the essentials necessary for understanding the special problems encountered in compiling this language.

ALDiSP provides all the constructs necessary for real-time programming (RTP). In particular, I/O transactions and time constraints can be specified. We consider the domain of real-time DSP the most demanding case of RTP: not only does it have all the problems of general DSP (number crunching on minimal, specialized target hardware platforms), it also has to cope with severe limits on timing.

In many respects, ALDiSP is an 'open' language: as defined, there is a 'basic kernel' that has to be provided by any implementation; a lot of additional data types (e.g., complex numbers and higher-dimensional matrices) and functions are defined, but do not have to be provided by every implementation, since they may not be needed in some of the possible application fields.

Conversely, each implementation will have to provide extra functions and objects not defined by the standard – namely those necessary for real-time I/O. Because it is impossible to define a uniform I/O interface that is optimally suited for all possible applications of ALDiSP, only the basic entities (synchronous and asynchronous I/O objects) and their behaviour are described in the standard; everything else is implementation-dependent.

Finally, it must be emphasized that not every ALDiSP program can be compiled for every target architecture: As the language allows for the specification of real-time behaviour, it is possible to write programs that need more computation power than can be provided by a given machine.

In theory, an ALDiSP compiler should verify that compiled programs conform to their timing specification; in practice, this is not done: Computing the timing behaviour needs a very precise machine model and can only be done for fully synchronous programs. For most practical programs, a simple test run should provide sufficient information as to whether or not a compiled program is fast enough.

0.2 Application Areas for ALDiSP

There are two more or less distinct areas where a language like ALDiSP can be employed:

- *Specification and Simulation*:
  During algorithm development, specifications are intended to be as succinct and precise as possible. At this stage of its life-cycle, the algorithm itself may still be little optimized and the simulation process is primarily concerned with correctness (especially concerning the numerical behaviour), not with speed. Signals are represented in floating point, so that rounding and cut-off errors need not be considered. Then, the cost/error tradeoffs of various fixed-point implementations are studied. To make this possible, the development system has to provide facilities for "bit-true" simulation of algorithms on the development hardware, usually some kind of workstation.

- *Code Generation*:
  Real-Time DSP applications are often implemented using "off-the-shelf" DSP hardware. These are processors adapted to a wide variety of DSP algorithms, but generally built around a MAC (multiply with a coefficient and add the result to an accumulator) datapath. Examples of these are the Motorola 5600x and 9600x and the Texas Instruments 320xx families of processors.

  When generating code for such processors, the output of a compiler must be of comparable quality to that a human programmer would write: As common C compilers do not generate code that meets

the speed requirements, most DSP programming is still done directly in assembly language, usually supported by large libraries of macro definitions[1].

The ALDiSP compiler, ac, is therefore targetted at low- to medium-throughput algorithms, as they are employed in typical telecommunications, audio, and control applications.[2] Such systems are often implemented on a basis of commercial DSP processors, or customized programmable cores.

The ac compiler can be used for both bit-true simulation and efficient compilation (which is, of course, the main goal of the compiler), because the central transformation and optimization phase of the compiler consists of a *partial evaluator* based on an *abstract interpreter*. The latter is, for all practical purposes, a full interpreter for ALDiSP. Enhanced with a user interface and a simulation environment that provides an I/O model, this interpreter can be used as the core of a simulation engine[3].

0.3 Overview of Compiler and Thesis

The compiler consists of the following phases, each of which is described in one or more chapters of the thesis:

- *Parsing.* An ALDiSP program is parsed into an abstract syntax tree (AST). No optimizations or simplifications are done at this level; AST syntax exactly mirrors the ALDiSP syntax. Chapter 2 describes the internals of the parser and how it interfaces with the following compilation stages.
- *Conversion into Lambda Form.* The Lambda Form (LF) is the intermediate representation employed throughout most of the following stages of compilation. LF is more general than the AST, and employs fewer syntactic forms. A formal semantics for LF is given in chapter 3; the semantics of ALDiSP itself is given indirectly by the transformation rules in chapter 4. These rules specify how an AST is translated into an LF program of equivalent semantics. The LF semantics employs a *state* that is passed around as a parameter of the evaluation functions. This state contains the current definitions of all references.
- *Simplification of the LF Program.* A set of semantics-preserving transformations is presented in the second half of chapter 4. Some of these transformation are applied to the initial LF program, others are used during various stages of the partial evaluation process.
- *Abstract Interpretation.* The abstract interpreter (AI) is a generalization of the standard interpreter, which in turn is the direct implementation of the formal semantics.[4] The AI executes the program on *abstract input*, and – together with an abstract scheduler (AS) – generates the set of all states that can be encountered when the program is run on actual data.
- *Re-writing the Program.* During its run, the AI has accumulated information about all function applications in a *call cache*; the AS has likewise created a graph that contains all possible states. Using this state graph as a skeleton, the program is "re-created". The call cache provides the information

[1] Due to the presence of aliasing problems (unrestricted pointers!), C is a language very ill-suited for the efficient compilation of DSP algorithms; single-assignment languages like SILAGE [110, 111], SISAL [84], or the functional subset of FORTRAN 90 are much better equipped for this task.

[2] Typical low-throughput algorithms are speech-processing tasks that have to cope with sampling rates of 8kHz or less, and 8 or 16 bit words. Typical medium-throughput algorithms work on high-quality sound signals; CD-quality sound needs 44.1 kHz, 16 bit words. Everything in the megahertz range is considered high-throughput.

[3] Such an interpreter would be too slow for practical simulation work, since it still has to lug around the extra load of internal book-keeping needed for code generation and recursion detection.

[4] Both "abstract interpretation" and "abstract interpreter" will be abbreviated as "AI"; this should not cause undue confusion.

necessary to create specialized function definitions. The program is re-written in a new intermediate representation, the *Code Form*.[5] Chapter 8 explains the Code Form and how it is generated.

- *Dumping the Program.* Finally, the Code Form program is transformed into the back-end format "M" that represents an abstract machine language. The state variables of the Code Form are absent from the M program; M uses only basic data types (integers and floating points); compound data structures are either atomized or globalized. A prototypical implementation of an M-to-C back-end (rather, a virtual M machine written in C) is used to test the M programs. Chapter 9 gives an overview of M and the M-to-C back-end.

The combination of abstract interpretation, abstract scheduling and re-writing is an instance of *Partial Evaluation* (PE). It makes up the majority of the compiler. Chapter 5 presents a general overview of PE techniques. Chapter 6 gives an introduction into abstract interpretation and explains the algorithms used for loop detection and generalization. Chapter 7 shows the integration of *state* into the AI, and how the abstract scheduler works.

0.4 Goals and Results

This work was undertaken with a number of goals in mind:

- There was the very practical need of having available a stable ALDiSP compiler that works correctly and adequately fast.[6] This goal was not be fully reached, since the current compiler is still too slow for interactive work.
- It should be shown that a language like ALDiSP – i.e., a functional language with many "inherently slow" features, a complex evaluation mechanism, and real-time constructs – can be compiled into efficient code. This goal was fully reached; all of the advanced language features can be removed, albeit at high costs in compilation resources and compiler complexity.
- The feasibility of partial evaluation as a fully automatic centerpiece of an optimizing compiler should be explored. The literature on AI and PE [17, 113] presents many cases where these techniques are used to realize very specific optimizations such as compile-time garbage collection, deforestation, strictness analysis, and numerical optimizations. Some of these are restricted in their applicability, others are not fully automated, i.e. need the support of additional annotations that guide the specializing process. It was found that partial evaluation is an enable technology for languages like ALDiSP, since it is probably impossible to compile a language with a complex semantics into efficient code *without* a large amount of compile-time reduction. Still, PE is no magic bullet: the design of the abstract domain directly mirrors the optimizations that can be achieved, and the common optimizations (inlining etc.) still have to be done, albeit at a later stage of the compiler.[7]

[5] This Code Form started out as a restricted Lambda Form, i.e. a proper syntactical and semantical subset of LF, but it underwent major modifications during the actual implementation of the code re-writing phase. One major difference between the Lambda Form and the Code Formis that the latter supports *explicit state*: state variables are threaded throughout Code Form expressions.

[6] A previous simulator, was written in C and Scheme. It did no pre-compilation, i.e. was written to implement a "direct semantics" for ALDiSP. The negative experiences made with this simulator – it was both buggy and horribly slow – served as impetus to introduce the Lambda Form that is central to ac.

[7] This might have been avoided by using a graph-based intermediate representation, but the ensuing tasks related to linearizing the graph would have solved the same problems in a different guise.

0.5 Related Work – Languages

In the industrial field, many DSP programs are still hand-crafted in assembly languages. Third-generation languages like C and ADA have been adapted to some DSP architectures by adding new keywords (e.g., memory layout directives) and/or restricting the languages by banning pointers, recursion, and sometimes even data-dependent loops. Many special-purpose DSP languages are described in literature [12, 54, 26, 108, 107]; most of them are based on the synchronous stream-processing model. There is also some overlap with hardware description languages [70, 85], since the synchronous data-flow model is also a natural description of clocked digital cicuits. Only few approaches are based upon asynchronous data-flow concepts [15]. Theoretical foundations in data-flow languages are based upon languages like Lucid [9] that have been developed independently from the DSP application area. One of today's most common DSP languages is SILAGE [110, 111] (and its commercialized successor, DFL). It is characteristic for a class of languages that offer simple translation into efficient code at the cost of reduced linguistic abstractions: no recursion, no overloaded or polymorphic function definitions, constant or constant-bound loop ranges, synchronous base model with up- and down-sampling extensions and "sampled asynchronicity" (asynchronous events modelled by synchronous streams using "no data" tokens).

0.6 Related Work – Partial Evaluation

Partial Evaluation (PE) is not a new invention; the term was coined in 1964 in the context of coping with "incomplete information" in LISP evaluation [81]. First PE-like optimizations can be traced back to the REFAL compiler [33] and Turchin's supercompiler [114, 115]; one theoretical milestone was the definition of the Futamura projections [52] (cf. section 5.3). Due to its origins in LISP evaluation, most early PE systems were based on the *on-line* principle, i.e. they reconstructed the program during its execution. An example is the REDFUN system [10, 56]. On-line PE is slow and error-prone; most of the later research went into the direction of *off-line* PE, which is based upon a few transformations (the two transformations 'fold' and 'unfold' are sufficient [22, 23]) that are guided by a *binding time analysis* (BTA) [67, 89, 29]. Off-line PE is vastly superior in terms of speed and simplicity of the partial evaluator itself; off-line systems also were the first to implement non-trivial self-application, which is an important theoretical goal. It is, however, not possible for off-line PE to achieve the level of optimizations that are possible in on-line PE. *Multivariant* off-line PE tries to reclaim some ground, but involves more complicated BTA schemes.[8] Current research is mostly concerned with improving the speed, robustness and precision of BTA schemes, most of which are based upon abstract interpretation.

0.7 Related Work – Partial Evaluation for Numerics and DSP

Berlin's articles [13, 14] are the first widely known references that describe the application of PE methods to the area of scientific programming. They describe a LISP framework in which PE is mainly employed to unroll loops and introduce software pipelining. The application is an n-body simulator that is partially applied to the known number and masses of the bodies.[9] By concentrating on the "inner loops" that are typical for numerically intensive programs, impressive performance increases are achieved.

Meyer [86] applies PE to an imperative language targeted at DSP applications. The language is Pascal-like, but restricted in the usual problem areas, namely recursion and pointer handling.

[8] The distinction between off-line and on-line PE blurs when multivariant off-line PE is considered; the BTA and multi-variant generation part of the off-line PE can become as complex and time-consuming as a full on-line PE!

[9] The n-body problem is to predict the vector and position of n masses at a time $t + k$, when their positions and vectors at time t are known.

0.8 Stylistic Questions, Thanks, &c.

> Never express yourself more clearly than you think.
> — Niels Bohr

I have tried to make this text as readable as possible. To this extent, many examples and explanatory footnotes are provided. I am indebted to the people that helped in proofreading (in alphabetic order: Guido Dunker, Andreas Fauth, Alois Knoll, Carsten Müller, Susan Wegner) for pointing out many typos, misleading details and taken-for-granted assumptions. Special thanks to Carsten for mentioning the A-normal form: without it, I would have re-invented a minor wheel another time; to Guido, for providing a "naïve reader"; to Andreas for patiently listening to my ramblings; and to Alois for the initial project idea and the "real-world" point of view. None of the remaining factual or stylistic errors are theirs.

This work was supported by an Ernst-von-Siemens stipend. Special thanks to Dr. Andreas von Zitzewitz for helping to provide this funding, and Dr. Gross for extending it another year.

Chapter 1

ALDiSP – *A Primer*

> A language that doesn't have everything is actually easier to program in than some that do.
> – Dennis M. Ritchie

In this chapter, the most important syntactic and semantic features of ALDiSP are described. An extensive suite of 'typical' example programs can be found in [45]. The original ALDiSP definition and report are [41] and [42].

1.1 Lexical Structure

An ALDiSP program is a single ASCII file. Comments start with "\\" and end with the "newline" symbol.

Preprocessor declarations start with a percent symbol ("`%`") in column 0 and include operator symbol syntax declarations like **`%INFIX`** and a file inclusion directive **`%INCLUDE`**.[1]

Identifiers are alphanumeric or 'symbolic' (strings consisting of [`+-*/$#%!&`]). An identifier in single quotes (`'`) loses its infix status; any sequence of characters between backquotes (`` ` ``) is a *symbolic constant* of undefined semantics.[2]

All identifiers are case-sensitive, while keywords are not.[3]

1.2 Overall Structure

An ALDiSP program consists of an arbitrary number of *sequential definitions*, followed by a *net list* (a recursive set of definitions), followed by an *initializing statement*:

... *definitions* ...
`net`
... *definition of a data flow net* ...
`in`
expression

[1] The `%INCLUDE` directive is new, i.e. not mentioned in [41, 42]; it turned out to be convenient for implementing libraries. Restricting parser declarations to start in column 0 makes it possible to use `%` as the modulo-operator.

[2] The concept of symbolic literals is also new. It is used by the systems implementor as a means of communication between library functions and the compiler/interpreter. For example, primitive functions are denoted by such literals.

[3] This feature is unique to ALDiSP, no other language known to the author behaves this way. While keywords can be written uppercase, lowercase, or mixed, all identifiers have to be written the way there were defined.

Evaluating the definitions that precede the `net` may not change the state; the net list and the initializing expressions can have side-effects.[4] This guarantees that the sequential definitions can be compiled independently.[5]

1.3 Evaluation Strategy

The evaluation of function applications is performed call-by-value; the arguments are evaluated left-to-right.[6] The special operator `delay` has the same functionality as the `delay` in Scheme [27]: it freezes the evaluation of its argument expression and creates a placeholder object (called a *promise*). While, in Scheme, a special primitive `force` is required to access the value of the promise (i.e., to initiate the evaluation), in ALDiSP the evaluation is started automatically the first time a strict primitive is applied to the promise[7]. Upon completion of the evaluation, the result replaces the promise.

1.4 Basic Types

ALDiSP provides a variety of built-in types and type constructors:

- signed and unsigned integers, floating point, fixed point, and complex numbers. All numeric types can be parameterized by size.[8]
- two different kinds of arrays: vectors (one-dimensional arrays) and matrices (two-dimensional arrays). Arrays may contain elements of arbitrary types, including other arrays. Vector literals are written as `#[`$x_1, \ldots, x_n$`]`.
- finite lists that may contain elements of arbitrary type. List literals are written as `#(`$x_1, \ldots, x_n$`)`. Basic list operations are "head", "tail", and "cons".
- infinite lists that may contain elements of arbitrary finite types. Input-driven infinite lists are called "pipes", output-driven infinite lists are "streams".

1.5 Auto-Mapping

When a function of type $(\alpha_1 \ldots \alpha_n \rightarrow \beta)$ is applied to an argument list of type (α_1 `list` $\ldots \alpha_n$ `list`), the function is automatically mapped to the lists (with type β `list` resulting). For this to work correctly, the lists must be of equal length.[9]

---

[4] Side-effects are changes to the state. The concept of "state" is explained in section 1.6. Examples for such changes are the creation of suspensions and all I/O operations.

[5] Such a separate compilation is not implemented in the current compiler.

[6] The alternative, call-by-need (lazy evaluation) was considered harmful, because the presence of side effects makes an explicit ordering mandatory. While it is possible to mix lazy evaluation and side effects, we consider such a mixture to be extremely confusing to the programmer. In addition, common wisdom is that strict (i.e., call-by-value) languages are easier to compile efficiently than lazy ones. Compilation of lazy languages tends to be based on a graph reduction model that is extremely unsuited for realization on DSP hardware, since it relies on the presence of a large heap and an efficient garbage collection scheme.

[7] Some Scheme implementations do so, since [27] allows it. It is, however, a nonstandard behaviour on which the programmer can't rely.

[8] All object "sizes" are measured in the number of bits used to represent an object.

[9] This restriction creates a small problem when infinite lists are considered: It is neither possible to check the length of such a list nor (in general) to prove that it is infinite. The semantics takes a practical stance: Each list that contains a suspension or promise in tail position is assumed to be an infinite pipe or stream. If, in the course of further evaluation, an end is encountered, this is considered a runtime error.

If only some arguments are lists (of equal length), then the non-list arguments are "expanded" to lists, i.e. (in LISP notation) `#(1 2 3)+100` evaluates to `#(101 102 103)`.

The same mechanism applies when the arguments are arrays (vectors or matrices); the arrays must be of equal size.

The original ALDiSP definition did provide for nested auto-mapping; i.e. things like

`#[#(1,2),3] + 10` $\rightarrow$`#[#(11,12),13]`.

This turned out to be too confusing; detecting programming errors became much harder because too many erroneous programs were accepted by the compiler.

1.6 Suspensions

The *suspension* is ALDiSP's sole all-encompassing real-time construct. It plays a central rôle in the language, enabling call-by-availability (input-driven computation), asynchronous event handling, and timing behaviour to be modelled.

An expression of the form

suspend *expr* **until** *cond* **within** *lower − limit* , *upper − limit* **end**

evaluates to a placeholder object (the suspension). When, after some time has elapsed, the *cond* becomes true (*cond*s are usually tests depending upon global state, tests for the evaluation status of other suspensions, or the literal `true`), the expression will be evaluated within the time range defined by the limits. The time range is taken relative to the point of time when *cond* becomes true; if it is `true` to start with, this is the time the suspension is created. After the evaluation of *expr*, the result replaces the placeholder (just like a promise is replaced by its value).

The evaluation model of the program as a whole is depicted in fig.1.1: The "current state" is made up from

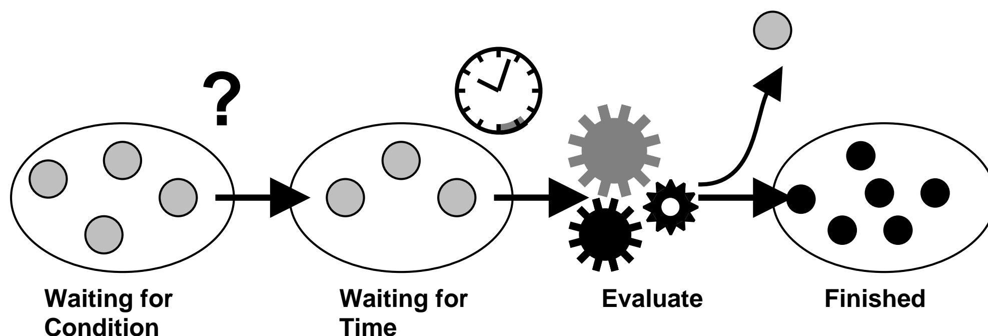

Figure 1.1: suspension state model

three pools containing suspensions, with one currently executing suspension. Newly created suspensions are placed in the "waiting" pool. When their *cond* becomes true, they are transferred to the "ready" pool; a timing counter is attached to each of the "ready" suspensions and initialized with an arbitrary point of time within the time range of its suspension.

When the expression of the current suspension is finished (a process in which many new suspensions may have been created and put into the "waiting" pool), the result is placed in the "result" pool, and some

arbitrary suspension whose timing counter is 0 is selected from the "ready" pool to be evaluated next. If there is no such suspension, the time is advanced, the counters of the waiting suspensions are decremented, and the conditions are tested.[10] Also, all I/O is performed during the time-advance. The whole program stops when the first two pools are empty and the evaluation of the current suspension terminates. Most real-time programs never terminate.

If this description seems to evince a very inefficient evaluation mechanism, it should be remembered that a suspension is roughly equivalent to an interrupt handler: creating the suspension installs an "interrupt vector", evaluating it means that the interrupt is raised and handled. Suspensions that wait upon I/O events can indeed be implemented as interrupt handlers.

1.7 *Blocking Access*

Whenever a strict primitive function accesses an unevaluated suspension, the "current process" is suspended until the data dependence can be resolved. While there is no such notion as "process" in ALDiSP, the net effect of cascaded blocking amounts to the same. This blocking access property of primitives can be modelled as a textual transformation:

prim(*obj*) → `suspend` *prim*(*obj*) `until is_available(`*obj*`) within 0.0 ms, 0.0 ms` *end*

Here, `is_available` is the primitive function that tests whether an object is accessible or an unevaluated suspension.[11] As an example for how this mechanism works, imagine the following program fragment:

```
let a = suspend f(x) until g() within 0.0 ms, 10.0 ms end
    b = (a+3)*17
    c = a-b
    d = 347
in
 c+d
end
```

This evaluates as follows: `a` evaluates to a suspension s_1; this shall be written `a` → s_1. The expression `a+3` evaluates to a suspension s_2 that waits for s_1 to complete; this shall be written `a+3` → s_2 ; s_2 ← s_1. The whole evalution proceeds as follows:

```
a → s1
a+3 = s1+3 → s2 ; s2 ← s1
(a+3)*17 = s2*17 → s3 ; s3 ← s2
b → s3
a-b = s1 - s3 → s4 ; s4 ← {s1 , s3}
c → s4
d → 347
c+d = s4 + 347 → s5 ; s5 ← s4
```

[10] The "virtual time advance" of the evaluation model is implemented as "waiting for the next clock tick". It depends upon the speed of the compiled program whether this virtual time can be as fast as the real time.

[11] In a way, `is_available` is the "non-deterministic" primitive - without it, there would be no way to implement the infamous "fair merge" function:

```
merge(sig1,sig2) =
    suspend if is_available(sig1) then hd(sig1) :: merge(sig2,tl(sig1))
                                  else hd(sig2) :: merge(sig1,tl(sig2))
            end
    until is_available(sig1) or is_available(sig2)
    within 0 ms, 0 ms end
```

The end result is a chain (to be precise, a directed acyclic graph) of suspensions. If the "first" suspension (s_1) is resolved, the rest up to s_5 will follow sequentially. This behaviour of "blocking chains" resembles that of a process.

1.8 Sequences

A *sequence* contains a list of declarations and/or expressions that are evaluated in sequential order. Its main use is in state-changing code, because there is a synchronization mechanism built in. If an expression in a sequence returns a suspension, the rest of the sequences is blocked until the suspension is resolved. An exemplary sequence is:

```
seq
  xxx_handle=open_file("xxx");
  x=read_from_file(xxx_handle);
  close_file(xxx_handle);
  x
endseq
```

The statements are evaluated sequentially; the result of the last expression is returned. The `close_file` line will probably evaluate to a suspension that is resolved when the file is closed.

1.9 Multiple Return Values

Functions may have more than one return value. Multiple return values are not "tuples" (which would be values of their own). Instead, each function can return any number of result values. A function that returns more than one value can only be used in a multiple variable definition.

That is, in the following example

```
func divmod(a,b) = { (a div b, a mod b) }
(x,y) = divmod(3,4)         \\ OK
'+'(divmod(3,4))            \\ ERROR ! NOT (3 div 4)+(3 mod 4)
xy = divmod(3,4)            \\ ERROR !
```

the last two lines are erroneous.

1.10 Conditionals

ALDiSP supports the traditional conditional construct (`if`). The "elsif" symbol is written as `[]`; the `else` clause is *not* optional, since `if` is an expression, not a statement. The syntax is

`if` $cond_1$ `then` $expr_1$
`[]` $cond_2$ `then` $expr_2$
⋮
`[]` $cond_n$ `then` $expr_n$
`else` $expr_0$
`endif`

The conditions are evaluated from top to bottom; if a condition evaluates to a suspended value, the whole expression behaves as a primitive function, i.e. forces an evaluation or blocks.

1.11 Local Definitions

The local definition construct (`let`) has the syntax

`let` $decl_1$ ... $decl_n$
 `in` *expr*
`end`

Declarations are evaluated sequentially from left to right. They cannot block; if in the course of the evaluatiuon of a declaration

`let` *var* = $expr_1$
 `in` $expr_2$
`end`

the $expr_1$ blocks, the *var* has as its value the resulting suspension. The expression $expr_2$ will only block if its evaluation contains a strict use of *var*.

1.12 Checking and Casting Types

ALDiSP supports a (run-time) type-checking model that allows verification and testing of arbitrary values and types. An expression

`[` *typ*`]` *expr*

is a *type declaration*, in which the programmer describes the type of the object that results from the evaluation of the expression. If the object is not of the declared type, a run-time error should be signalled (if an interpretation/simulation is in progress) or the program may behave undefined (if the program is compiled with the respective checks turned off). Type checking can also be used to reduce the domain of functions by giving them a restricted type.

`[!`*typ*`]`*expr*

is a *type cast* or *conversion*. E.g, `[!Real]42` converts the integer `42` to the "semantically equivalent" floating point representation. Additional conversion functions may be installed by the user by declaring the overloaded function `cast_general`[12] with one argument.

Checks and casts are strict: if a type check/cast is applied to a promise, it is forced; if it is applied to a suspension, the whole operation blocks.[13]

1.13 Exceptions

Like many other programming languages, ALDiSP provides constructs to cope with exceptional behaviour. In addition to this purpose, exceptions are used to model locally deviating numerical behaviour. This means

[12]There exists quite a notational confusion in this area; it is by no means obvious what the differences between "conversion", "coercion", and "casting" are. Usually, "casting" is the re-typing of a value by giving its representation a new interpretation, while "conversion"/"coercing" actually implements some kind of value-preserving function that changes the representation of a value and its type.

[13]As an exception, checking and casting against "`Obj`", i.e. the type of all possible objects, is a no-op.

that not only error conditions (division by zero etc.) are handled by exceptions, but that the same language construct also handles rounding modes and overflow handling.[14] For example,

```
guard a+b/c
on Round(x,n) = x
on AdditionOverflow(x) = {writeToPort(stderr,"Overflow!");
                          return(MaxInt)}
end
```

evaluates `a+b/c` with a 'cutoff' rounding mode and a signalling overflow handler.[15] While the `Round` exception returns to the place where it was called, the `Overflow` exception aborts the evaluation of `a+b/c` and returns `MaxInt` as the total result of the `guard` form.[16]

There is no special syntax for raising an exception; it is called like any other function.

1.14 Lists, Streams, Pipes

The basic abstraction in any DSP-oriented language is the *signal*: "synchronous" languages like Silage[17] [110, 111] use "current sample" and "k-th previous sample" signal variables; ALDiSP uses the *list* abstraction: the head of the list is the "current sample", the following element is the "next" one, and so forth.

As lists are usually finite objects, some imagination is necessary to understand the notion of lazy infinite lists.

ALDiSP has borrowed from other functional languages the *stream* approach to implement infinite lists: a stream is a "potentially infinite" list whose first k elements are explicitly present; the rest are modelled by a generator function that can deliver any number of additional elements.
A simple example is a sine generator:

```
func sinus(n,delta) =
 let newval = n+delta
 in
  sin(n) :: sinus(if newval>=2*Pi then newval-2*Pi else newval end,
                  delta)
 end
sin0 = sinus(0,0.01)
```

Here '`::`' is a lazy list constructor, with its second argument `delay`ed. The operator '`::`' is defined in terms of `delay` and the primitive list constructor `cons`:

```
%R-INFIX '::'                  \\ infix, right associative
macro a::b = cons(a,delay b)
```

[14] Rounding and overflow handling is of paramount importance in DSP, especially in maximizing output accuracy of the system. One of the most frequently used modes (besides simple cut-off) is "saturation" arithmetic, where the deleterious effects of an overflow are reduced by causing the output value to saturate at the maximum positive/negative value. The ALDiSP library also provides "round to zero", "round to even", etc. To our knowledge, ALDiSP is the first language that uses exceptions to implement different rounding modes. Usually, some global state parameters or parameterized numeric functions are needed to model different behaviours.

[15] `Round` is the predefined exception to be called when a result cannot be represented exactly within the specified precision: its arguments are the expressible part of the result number plus the first following bit. In the example, this bit (`n`) is simply ignored.

[16] There is a change from the original ALDiSP definition where `return` marked the returning case and the jump back to the `guard` was the default.

[17] Version 2.0 of Silage handles multi-rate and asynchronous systems by introducing explicit "upsampling" and "downsampling" facilities.

It is necessary to define '`::`' as a macro because function calls are by value. A call-by-value operator would of course defeat the whole idea of a lazy datatype constructor![18]

For asynchronous real-time applications, streams provide insufficient expressive power: input- (or data-) driven applications cannot be implemented using them[19], so *pipes* were introduced. Pipes are input-driven streams implemented as chains of suspensions. For example,

```
proc pipeFromRegister(reg,interval) =
  suspend cons(readFromRegister(reg)
               pipeFromRegister(reg,interval))
  until true
  within interval,interval end

net
  input = pipeFromRegister(reg_1,0.2 ms)
  output= some_filter(input)
in
 writeToRegister(reg_2,output)
```

`pipeFromRegister` delivers a pipe generated by sampling the contents of a register `reg_1` at defined points in time; some filter function is then applied to this stream before it is written back to an output register `reg_2`.

Due to the blocking properties of pipes (which are basically a consequence of the blocking properties of the suspensions that implement them), they can be used both syntactically and semantically like streams. This similarity makes it possible for the programmer to subsitute stream-generating functions for real-time inputs, which can be very convenient when testing algorithms.

1.15 Declarations

Declarations can be used to introduce arbitrary values, abstract datatypes, exceptions, new functions, and general types:

```
max_int = 2**16-1
abstype Nat = Null | Succ(Pred:Nat)
abstype ListOf(x) = Empty | Cons(Car:x,Cdr:ListOf(x))
exception Overflow(x) = {error("overflow"); max_int}
func multOf(n)(x) = Int(x) and x mod n == 0
type multOf3 = multOf(3)
```

The last lines of the example present one of the most unique (and irritating to implement) features of ALDiSP: Just as every type can serve as the predicate that detects the objects of this type (this is used in the `Int(x)` expression), any predicate can be used as a type.

Combined with the type declaration construct, this duality makes it possible to introduce arbitrary assertions into the program. These can be used both during simulation as a testing tool and during compilation as a source of hints to the compiler.[20]

[18] Indeed, macros were only introduced to make it possible to abstract from non-strict expressions.

[19] They can, but only with one of the following 'hacks':

- special "non-input" values can be provided – this amounts to busy waiting,
- an explicit global scheduler that emits a stream of "there is input on port x" information can be installed

Both approaches cannot provide acceptable performance in systems consisting of many asynchronous signal sources that are triggered with a low probability. A typical example for such a system is a telephone exchange.

[20] The current implementation does not utilize type annotations in this way.

1.16 Modules

ALDiSP supports a simple module facility as illustrated by the following example:

```
module xx
  export a : Int -> Int,
         b,
         c as d
  func a(n) = n+1
  b = 42
  c = a
endmodule xx

import a : Card -> Card from xx
import b as bb from xx
y = xx.d(bb)           \\ y = 43
```

When an object are imported and exported, it can be given a new name, or it can be re-typed. Re-typing amounts to type checking using the [*type*] operator.

1.17 Functions

We have already seen how function symbols can be defined as infix-operators, which is a purely syntactic feature.

Functions can also be *overloaded*:

```
import '+','-','*','/' from Integers
overloaded import '+','-','*','/' from Floats
import overloaded '*', PointInfinity from Complex
```

An "`overloaded import`" overloads all values that are imported, while an "`overloaded`" marker in the import list overloads only the immediately following object.

Overloaded functions need not have the same arity (number of parameters):

```
func map(f,a) =                                      \\ map/2
  let func loop(a) = if a is null then null
                     else f(head a) :: loop(tail a) end
  in loop(a)
  end
overloaded func map(f,a,b) =                         \\ map/3
  let func loop(a,b) =
        if a is null or b is null
           then null
           else f(head a,head b) :: loop(tail a,tail b) end
  in loop(a,b)
  end
```

The code shown above uses the predefined operator `is` to test whether its first argument was generated using the datatype constructor that is its second argument. `head` and `tail` are the usual list selectors.

Overloaded functions are nonrecursive. As a consequence, `map/3` has to be defined in terms of an ancillary `loop` function, because the only `map` visible in the scope of `map/3` is the "old" `map/2`.[21]

[21] I had defined a semantics for overloaded recursive functions, but it got too complicated for practical use. The restriction shouldn't cause any practical problems.

Chapter 2

The Parser

A formal parsing algorithm should not always be used.
– D. Gries

Implementing the front-end (scanner and parser) of a compiler is usually considered a more or less trivial task. Looking at the issue in greater depth, though, scanners and parsers are found to be full of details that tend to be managed by introducing ad-hoc solutions: the scanner – and, at the other end of the compilation trajectory, the code generator – is one of the few parts of a compiler that has to interact directly with the operating system; scanner and parser have to provide essential information for error diagnostics; many languages have "dark corners" in their lexical or syntactical structure that make trouble when using automatic front-end generators like `lex` and `yacc`; lastly, those tools often tend not to be rich enough in their functionality so that hacks are needed to reach the desired goals.

The development of the parser described here was influenced by the first ALDiSP parser that was developed in 1990 using `lex` and `yacc` in C. That parser created a textual output representing the syntax tree in LISP-like notation; for example,

```
net
in
 a(if b(c,d) then e(f,g)
   [] h then 42
        else 99
   endif)
endnet
```

was transformed into

```
(PROGRAM (())
    (LOCAL ()
(APPLY (ID "a")
    (IF ((APPLY (ID "b") (ID "c") (ID"d")))
((APPLY (ID "e") (ID "f") (ID "g")))
((IF ((ID "h"))
     ((CONST 42))
     ((CONST 99))))))))
```

This tree was used as input to an ALDiSP-simulator[1] written in Scheme [27].

[1] This simulator was a very complex affair because the abstract syntax was *not* simplified prior to simulation – the goal was to write a 'direct' ALDiSP evaluator. The simulator worked, and was used to test the interaction of the basic language features, but its slowness rendered it impractical for interesting programs. When using it, Turner's apocryphal comment about an early version of Miranda executing "with the speed of continental drift" came to mind.

In the next sections, this first parser will be mentioned whenever a design decision was made because of experiences with it.

2.1 Overview

The `ac` compiler is implemented entirely in Standard ML [87] and has thus a well-defined behaviour – except in the area of I/O, which is system-dependent.[2]

The front-end was developed using the **sml-lex** and **sml-yacc** tools; these have specification languages similar to their C counterparts, but their implementation interfaces are quite different. UNIX `lex` and `yacc` generate two main functions, **yylex** and **yyparse**, which deliver, respectively, the next token (coded as a small integer) or the next parse result. Their SML counterparts communicate via token streams. A great deal of time was spent in learning how to manage this interface; especially, how to implement the preprocessor as a filter function for these streams.[3]

The parser consists of three parts: lexical analysis, preprocessing, and parsing. A symbol table is maintained by the preprocessing phase, which also handles the preprocessor declarations (these are the fixity declarations (`infix`, `prefix` and `postfix`) and the file inclusion directive). An overview is given in figure 2.1.

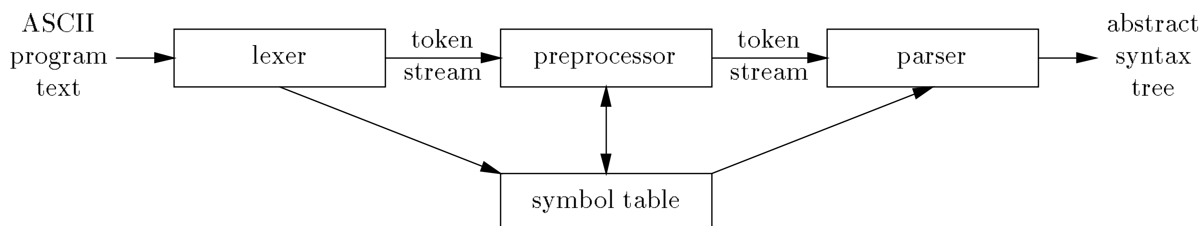

Figure 2.1: Parser Submodules

2.2 Lexical Analysis

Lexical analysis is trivial; preprocessing declarations like **%infix** are tokenized, but otherwise ignored. The resulting token data type is

```
datatype lexresult =
     STRING of string
   | CONST of string
   | FLOAT of real
   | INT of int
   | ID of string
   | QID of string
   | SEP of string
   | ELIF | TYPDECL | VECTORCONST | LISTCONST
   | INFIX | PREFIX | POSTFIX | RINFIX | INCLUDE
   | IGNORED | NEWLINE | EOF
```

[2] The implementation used is SML/NJ; those few additions to the standard that it offers – and are used by `ac` – concern the array interface and conform with an unofficial agreement of all major SML implementors today. `ac` should therefore be perfectly portable.

[3] The designers of `sml-lex` and `sml-yacc` seem to have missed the necessity of putting a filter between these two tools, and therefore forgot to better document this interface. Some experimentation was needed to create a type-safe solution that did not violate the complex module interface.

```
%term INFIX_L1 of Symbol.T | ... | INFIX_R5 of Symbol.T
    | PREFIX  of Symbol.T | POSTFIX of Symbol.T
    | ID of Symbol.T
    | CONST of string
    | STRING of string
    | INT of int
    | FLOAT of real
    | NET | IN | ENDNET | LET | WHERE | ENDLET | MODULE
    | ENDMODULE | IF | THEN | ELSE | ELIF | ENDIF | DELAY | SUSPEND
    | UNTIL | WITHIN | ENDSUSPEND | IMPORT | FROM | AS | EXPORT
    | GUARD | ON | ENDGUARD | SEQ | ENDSEQ | RETURN | VAL
    | FUNC | PROC | TYPE | EXCEPTION | ABSTYPE | OVERLOADED
    | MACRO | REC | END | ENDREC | TYPCAST | VECTORCONST
    | LISTCONST | TYPDECL | ARROW | BRACKCL | PARCL
    | BRACEOP | BRACECL | BAR | PAROP | COMMA | EQUAL
    | COLON | DOT | SEMI | EOF
```

Figure 2.2: Parser Input Token Type

All separators (comma, semicolon, etc.) are represented as `SEP` tokens. Tabs and spaces are absorbed. Newlines are passed along as tokens because they might terminate lines that contain preprocessing declarations.

A `QID` is an identifier enclosed in single quotes, which dissociates it from any fixity attribute. The token stream that enters the preprocessor knows no tokens that represent keywords, as keywords are recognized during preprocessing. A `CONST` is a 'symbolic constant', i.e. a string enclosed in backquotes.

Any input line that starts with a '`%`' which is *not* followed by one of the known preprocessor declaration keywords is ignored; an `IGNORED` token is passed along instead, and the rest of the line is skipped. The lexer does not generate error messages; only the preprocessor and parser do so.

2.3 *Preprocessing*

The main function of the preprocessor is the analysis of preprocessor declarations, of which there are currently two kinds: textual file inclusion (`%include`) and symbol fixity declarations (`%infix`, `%prefix`, `%postfix`). The `%include` directive is easy to implement: The preprocessor finds, opens and closes all files; it calls the lexer with the opened file handles and transforms the resulting token stream into the token stream suitable for the `sml-yacc` parser. To include another file, a new incarnation of the scanner is applied to a character stream representing the included file; the resulting token stream is then inserted into the original token stream, replacing the `%include` directive.

The fixity declarations define the associativity and precedence of tokens that can be used as unary and binary operators. The `sml-yacc`-generated parser needs distinct tokens for the separators and the keywords; it cannot cope with literal strings like standard C `yacc` can. The type of tokens (cf. fig. 2.2) suitable as input to the parser is generated by the following rules:

- `SEP(",")` ⟶ `COMMA`, `SEP(";")` ⟶ `SEMI` etc., for all "punctuation" characters.
- `ID("net")` ⟶ `NET`, `ID("if")` ⟶ `IF` etc.; these string comparisons are case-insensitive (cf. chapter 1.1).

- `ID(`x`)` $\longrightarrow$ `ID(Symbol.intern(`x`))`, if there is no fixity declaration for x. The structure `Symbol` implements the global symbol table[4]; `Symbol.intern` enters a string into the table.
- `ID(`x`)` $\longrightarrow$ `INFIX_L3(`x`)`, if the preprocessor has encountered a fixity declaration (in this case left-associative, binding strength 3) for x.
- Fixity declarations are parsed 'by hand'. Error-handling is unsophisticated; in case of an unexpected token, an error message is emitted, and everything up to the next `NEWLINE` is discarded. The fixity information is stored in the symbol table.
- A few tokens are passed along as they are: `TYPDECL` stands for "`[`", `TYPCAST` for "`[!`", `VECTORCONST` for "`#[`" and `LISTCONST` for "`#(`".

The handling of preprocessing directives does not employ any formal method; it is written as a functional automaton, i.e. a set of tail recursive functions. The standard action is to copy input to output, element-wise applying the transformations. Only when encountering a preprocessing directive, the arguments are parsed upto the next newline. The fixity declarations are stored as flags in the symbol table.

2.4 The Parser

The parser directly models the grammar as it is presented in [41] and [42]. The grammar is alse reprinted in appendix B. Each nonterminal rule returns an abstract syntax tree (AST) expression. The abstract syntax, modelled as a set of recursively defined data types, is given in fig. 2.3 in the form of an SML data type declaration.[5]

The parser converts the concrete syntax into the abstract syntax without performing any significant transformations. There is only one nontrivial transformation performed by the parser, which is connectedd with the treatment of typed infix function declarations.

When a function declaration is parsed that is written in infix form and contains an explicit typing annotation, the parser is in trouble. The abstract syntax for function declarations mirrors the concrete syntax of the standard (non-infix) function declaration, which has the general form

[`overloaded`] `func` *name*(*arg₁*:*typ₁*, ...) ...:*typ₀*=*expr*

Note that typ_0 is the result type of the function, i.e. the type of values that result from evaluating *expr*. The generated term models this form directly:

`Aparamdecl(pos,isOverloaded,Afunc,[[(`arg_1`,`typ_1`)...]...],name,`*expr*`)`

In contrast, an infix declaration has the form

[`overloaded`] arg_1 `op` arg_2 `:` *typ* `=` *expr*

Notice that *typ* denotes the type of the function as a whole. The problem results from the choice of distinct argument and result type expressions in the abstract syntax: how can the separated types $typ_0 \ldots typ_n$ be generated from this one type expression? After all, *typ* can be an arbitrary expression!

[4] This table is a very non-functional thing; but passing it along as an extra parameter would have complicated the implementation of the parser, without gaining much in abstraction.

[5] For those unfamiliar with SML, a short example: `datatype A = B(C) | D` declares a type `A` to have two constructors. `B` is a unary constructor with an argument of type `C`, while `D` is a nullary constructor.

```
datatype Aprogram =
  Aprogram of Adecl list * Adecl list * Aexpr
and Adecl
  = Asimpledecl of pos * bool * id * Aexpr
  | Aparamdecl of pos * bool * Aheader * Aparam list list * id * Aexpr
  | Amultidecl of pos * id list * Aexpr
  | Amoduledecl of pos * id * Aexport list * Adecl list
  | Aimportdecl of pos * id * Aexport list
  | Aabstypedecl of pos * Aparam list list * id * Aconstructor list
  | Arecdecl of pos * Adecl list
and Aparam
  = Aparam of id * Atype
and Aexport
  = Aexport of pos * bool * id * id * Atype
and Atype
  = Afunctype of Atype list * Atype list
  | Aexprtype of Aexpr
and Aexpr
  = Aapp of pos * Aexpr list
  | Aseq of pos * Astmt list
  | Atupel of pos * Aexpr list
  | Acond of pos * Aexpr * Aexpr * Aexpr
  | Avar of pos * id
  | Asymbolic of pos * string
  | Aconst of pos * string
  | Aint of pos * IntRepr.T
  | Afloat of pos * FloatRepr.T
  | Astring of pos * string
  | Acast of pos * Atype * Aexpr
  | Acheck of pos * Atype * Aexpr
  | Adelay of pos * Aexpr
  | Asuspend of pos * Aexpr * Aexpr * Aexpr * Aexpr
  | Aguard of pos * Aexpr * Adecl list
  | Alocal of pos * Aexpr * Adecl list
  | Areturn of pos * Aexpr list
and Astmt
  = Aexpr of Aexpr
  | Adecl of Adecl
and Aconstructor
  = Aconstructor of id * Aparam list
and Aheader
  = Afunc | Aproc | Atype | Aexception | Amacro
```

Figure 2.3: The AST Data Type

The solution as implemented by the parser is somewhat inelegant, but correct. The parser generates the abstract equivalent of the following expression:

```
[overloaded] op =
   let func tmp(arg1, arg2)= expr
   in [typ]tmp
   end
```

This is correct since the semantics of an 'external' type declaration (`[typ]`expr) is the same as that of an internal one (`func` name:type=expr).[6]

2.5 Blind Alleys

During the development of the front-end, quite a few design errors were made, detected, and corrected. Some of these were detected later, when scanner and parser had to be integrated into the compiler's framework.

2.5.1 Representation of Literals

How are literals (especially numbers) represented internally? This question seems trivial, but it has far-reaching consequences, since many phases of a compiler have to deal with literal values.

In most compilers, literals are transformed to an internal representation in the lexer: strings stay strings, numbers are transformed to machine form, booleans are transformed into references to global variables, enumeration literals or numeric representations. Especially when semantic actions are performed on the parse tree, it is handy to have numbers available in a format that corresponds to the one used within the compiler itself.

This approach leads to problems if a cross-compiler is written (as is the case with `ac`). On any given machine, most programming languages support the same numeric formats – those directly supported by the hardware. Usually it is only in cross-compilers that the implementation language provides a numeric format differing from that of the object language.

ALDiSP allows very large and precise numbers; declarations like

```
large_const = 74356724398652743864832157834653854365248735 64
pi = 3.14159265358979323846264338327950288419716939937 51
```

are allowed and well-defined. If only those numeric representations were used that are provided by the language the compiler is implemented in (in this case, SML/NJ), precision would be restricted to 32-bit integers and standard (IEEE) floating-point numbers.

Alternatively, literals can be represented as strings, i.e. the way they were written in the program. This strategy, while being "safe", turns out to be quite annoying when transformation rules need to be implemented: when a lot of integer constants are created and parsed in the transformation phase, calls to `stringToInt` and `intToString` crop up everywhere. Both for efficiency and legibility reasons, this is an unsatisfactory solution.

In `ac`, the right way turns out to be the usage of the abstract interpreter's semantic value representation (cf. chapter 3.1.2). The abstract interpreter (AI) implements a generalization of the standard semantics, and provides the back-bone for code generation. Because the AI has to be able to deal with these values anyway,[7] it will know how to represent literals and how to apply the primitive operators to them.

---

[6] An untyped declaration is the same as a declaration of `Obj` (the type of all objects).

[7] The AI should at least be able to do correct constant folding, for which it has to be able to represent all values that can occur in literals!

In the current implementation, the semantic values defined in the AI do not directly represent multi-precision integer and fixpoint numbers. Instead, the employ value data type defined by separate libraries (one each for integer and floating-point multi-precision values). The parser passes the input strings to these libraries, which transform them into their internal format. The resultant values are stored in the parse tree.[8]

2.5.2 *Implementing The Preprocessor*

The first parser had no separate preprocessor; preprocessing was done using **lex** states. This approach proved to be complicated. The essential part was quite hard to understand (cf. fig. 2.4).

```
^"%"    {BEGIN parserline;}
<parserline>"INFIX"{iws}   |
<parserline>"L-INFIX"{iws} {base_type=BINOPL3;cnt_fix=0;BEGIN firstname;}
<parserline>"R-INFIX"{iws} {base_type=BINOPR3;cnt_fix=0;BEGIN firstname;}
<parserline>"PREFIX"{iws}  {base_type=PREOP  ;cnt_fix=0;BEGIN firstname;}
<parserline>"POSTFIX"{iws} {base_type=POSTOP ;cnt_fix=0;BEGIN firstname;}
<parserline>"INCLUDE"{iws} {BEGIN includefile;}
<includefile>[^ \n\t]*    {strncpy(filenamebuffer,yytext,yyleng);
    filenamebuffer[yyleng]='\0';
                           includeFile(filenamebuffer);
    BEGIN 0;}
<firstname,nextname>{id}   {fixes[cnt_fix++]=intern(yytext,yyleng);
    BEGIN nextname2;}
<firstname,nextname>\'[^']*\'{fixes[cnt_fix++]=intern(yytext+1,yyleng-2);
      BEGIN nextname2;}
<firstname,nextname>\"[^"]*\"{fixes[cnt_fix++]=intern(yytext+1,yyleng-2);
      BEGIN nextname2;}
<nextname2>","    {BEGIN nextname;}
<nextname>[1-5]    {if (base_type==PREOP || base_type==POSTOP){
      yyerror("No precedence for Pre- & Postfix!");
    } else {base_type+=yytext[0]-'3';}
      {int i; for(i=0;i<cnt_fix;i++)
              {checkToken(fixes[i],base_type);
        token2type(fixes[i])=base_type;}}
    BEGIN 0;}
<nextname2>{ioptws}\n    {{int i;
    for(i=0;i<cnt_fix;i++)
      {checkToken(fixes[i],base_type);
       token2type(fixes[i])=base_type;}}
    BEGIN 0;}
```

Figure 2.4: An early combined lexer/preprocessor

The current implementation of **ac** employs a separate preprocessor, which is a filter between the lexical analyzer and the "real" parser. This approach turns out to be more flexible and easier on modifications. As an extra benefit, the preprocessor can handle the keyword recognition task, which earlier was done by

[8] In an earlier version, AI semantic values were generated and incorporated into the AST. This incurred a problem of modularity: since function closures are amongst the values modelled by the AI value data type, it has to have access to the Lambda Form data structures. These, in turn, are generated from the abstract syntax tree. Since SML does not allow recursive module definitions, this cyclic dependency had to be broken. One possible solution was to implement closures as (integer) tags to a global closure table, thus voiding the need for explicit references to the Lambda Form type. The separation which is now employed is more elegant, since it employs a hierarchy of modules: at the lowest level are the modules that provide for literal values and operations on them; the AST data type uses these values, as does the AI value data type.

priming the symbol table – a method which was complicated by the case-insensitive comparison for keywords, because the knowledge about this oddity had to be buried in the `intern` function of the symbol table.

2.5.3 Error Handling

The first parser had extra grammar rules to support error recovery; e.g. instead of the simple grammar fragment

```
if   : IF exprs THEN stmts else ENDIF ;
else : ELSE stmts
     | ELIF exprs THEN stmts else
     ;
```

some error productions were introduced.

```
if  : IF exprsTHEN stmts elseENDIF ;
exprsTHEN: exprs THEN
 | error THEN
 ;
elseENDIF: ELSE stmts ENDIF
 | ELSE error ENDIF
 | ENDIF
 | ELIF stmts THEN stmts elseENDIF
         ;
```

Theses error rules bloated the whole grammar and made it harder to understand and modify.

Luckily, `sml-yacc` supports its own error-detection mechanism that is at least as good as the hand-written error rules used in the first parser. The grammar could therefore be implemented directly without any changes.

2.5.4 Line Numbers and File Names

The first abstract syntax tree did not contain any references to line numbers. First experiences with the abstract interpreter showed that it was quite a chore to locate errors in the source program.

Surprisingly, it took only two days to add line numbers to the parser and all subsequent modules (at that time, CPS conversion and abstract interpretation). Each node of the AST has now an additional attribute of type `Pos.T`, which contains two line numbers that name the "span" of a construct, and the source file name. The latter is necessary since practical ALDiSP programs tend to `%include` many library files, directly and indirectly.

The biggest practical implementation problem was the inability to access a "current line number" of non-terminals within `sml-yacc` semantic rules. It was known that the parser keeps track of positions, but it was not documented how they can be accessed. Some poking in the generated code showed what variable names were generated by `sml-yacc`. To pick an arbitrary example, the `typCast` rule is now defined as:

```
YtypCast: TYPCAST Ytype ECKZU YcExpr
( A.Acast(mkPos(TYPCAST1left,YcExpr1right), Ytype,YcExpr) )
```

The first line represents the grammar rule

$$typCast \quad \rightarrow \quad \texttt{\#[}\ type\ \texttt{]}\ Expr_{closed}$$

The second line contains the SML code that creates an `Acast` node out of three arguments. The second and third arguments are the cast's type and expression; the first argument represents the current position.

The constructor `mkPos` takes two line numbers as arguments, these are the line at which the "`[`" starts (`TYPCAST1left`) and the line at which the `YcExpr` ends (`YcExpr1right`). The `mkPos` constructor takes the current file name from a global variable.

2.5.5 CPS Intermediate Form

In the beginning, the parser was intended to create code in *continuation-passing style* (CPS). All necessary simplification should be done in the parser, i.e. the CPS-transformation should be distributed over the semantic actions of the grammar rules.[9]

This approach seemed to have quite a few advantages: At that stage, I thought that a CPS form would be the only intermediate representation throughout all symbolic evaluation and code generation. Because each new data structure requires its own debugging support (at least a print function), I planned to use only one data representation, which was to be generated directly in the parser. The alternative – which is now used – leads to the creation of a whole data type "ALDiSP AST" that is used only for communication between the parser and the succeeding module. That module's only task is to simplifying the AST and it into Lambda Form; it is described in chapter 4.

CPS is a form of λ-calculus in which functions never return to their call site. Instead, each function is given an explicit continuation, which is called with the result of the function. As an output language for a parser, CPS has distinct advantages:

- *continuations:*
  Using a CPS form allows to explicitly represent nearly all possible kinds of control flow; especially, the continuation capture needed to model the behaviour of the different exception handling and exception raising scenarios.[10]
- *multiple outputs:*
  Using a CPS, many-valued functions can be represented easily: where usually a continuation function of one argument is called, an n-ary continuation can also be used. This approach to handling multiple return values avoids introducing a "tuple-type" if it is not present in the language itself.

Implementing the transformation rules in the parser is straightforward if they can be specified as attribute-grammar rules with recursive-descent evaluation order, since such attribute grammars can be implemented very nicely in SML using the "named struct" facilities offered by the language. For example, a set of equations

$$\begin{aligned}
\texttt{expr} &= \texttt{app}(\mathit{expr_1} \ldots\ \mathit{expr_n})(\downarrow\ \mathit{def}, \uparrow\ \mathit{use}, \uparrow\ \mathit{tree}) \\
\mathit{def} &= \downarrow\ \mathit{def} \\
\mathit{use} &= \mathit{expr_1}.\mathit{use}\ \cup\ \ldots\ \cup\ \mathit{expr_n}.\mathit{use} \\
\mathit{tree} &= \texttt{app_node}(\mathit{expr_1}.\mathit{tree}, \ldots\ \mathit{expr_n}.\mathit{tree})
\end{aligned}$$

with inherited attributes *def* and synthesized attributes *use* and *tree* can be implemented using a function that takes the inherited attributes and returns the synthesized attributes (each wrapped up in a data structure):

[9] There are many possibilities to specify and implement a CPS transformation; cf. [48] for a discussion.

[10] The exception handler (a `guard` expression) has to capture the "current continuation", which is called when an exception is raised and `returns`.

```
app : expr '(' exprs ')' {
  fn (IN:{def:SymbolSet}) =>
  let val func_out = $expr IN
      val args_out = map $exprs IN
  in
  {use  = reduce union (map #use (func_out :: args_out)),
   tree = app((#tree func_out) :: (map #tree args_out))
  }
  end
}
```

In this code, `$expr` is the value of the `expr` that was matched by the parser. This value is itself an attribute-transforming function. Applying the `$expr` and `$exprs` functions to `IN` amounts to "passing" the inherited attributes "down"; `func_out` and `args_out` are the resulting (bundles of) synthesized attributes that were "passed up". Using `#use` and `#tree`, specific attributes can be selected from these bundles.

To implement a CPS transformation, only a single extra attribute is needed:

the current continuation expression, usually a simple variable. But naïve CPS transformation algorithms generate bloated code. A trivial expression like

```
3*a
```

might be transformed to

```
((lambda (tmp_1)              ; tmp_1 = *
  ((lambda (tmp_2)            ; tmp_2 = 3
    ((lambda (tmp_3)          ; tmp_3 = a
      (tmp_1 tmp_2 tmp_3 K)
     a)
   3)
 *)
```

(where `K` is the 'surrounding' continuation). A more sophisticated algorithm knows the difference between "atomic" and "non-atomic" expressions (and generates `(* 3 a K)`, as it should be). To implement such a more refined CPS conversion on a 'distributed' level (while parsing is done), a new data type `AtomicOrComplex` must be introduced – and the code for the transformation rules doubles (in some cases even quadruples) in size.

2.5.6 *CPS Transformation: The Morale*

The CPS transformer worked; and it was buggy. The generated output was too big to check by manual inspection.

The generation of source code for the parser with `sml-yacc` was quite fast; it took less than a minute to compile the parser specification. Compilation time for the generated parser took about 8 minutes (Sun Sparc II, 32Mb). The modify-compile-test cycle was therefore in the order of 10 minutes.

Hence the first parser was a complex, incorrect piece of software that took a long time to compile and that was very hard to test.

After two weeks of 10-minute turn-around times, a complete re-write was done in three days: one day for the data structure and support functions, one day for the parser, and one day for implementing a compilation into Lambda Form. A LF-to-CPS transformer took an additional three days.

The major morale is therefore: one should try to keep the compile-link-test cycle as low as possible, and split the modules accordingly.

(CPS as an intermediate form was later abandoned.)

Chapter 3

The Lambda Form

"...Then anyone who leaves behind him a written manual, and likewise anyone who receives it, in the belief that such writing will be clear and certain, must be exceedingly simple-minded..."
– Plato, *Phaedrus*

This chapter and the following one formally define the semantics of ALDiSP programs. Since ALDiSP is much too complex to describe in a "direct" way, an intermediate representation, the *Lambda Form* (LF), is introduced. This chapter describes the semantics of LF programs; the next chapter shows the set of functions that transform abstract ALDiSP programs, as generated by the parser, into LF programs.

The syntactic[1] definition of LF programs is given by a set of SML data-type declarations:

```
datatype Lprog = Lprog of Ldecl * Lexpr

datatype Ldecl
  = Ldecl of bool * Var * Lexpr
  | Lpard of Ldecl list
  | Lseqd of Ldecl list
  | Lfixd of Ldecl list

datatype Lexpr
  = Lambda  of int * (Var * Lexpr) list * Lexpr
  | Lvar    of Var
  | Lit     of SemVals.T
  | Lapp    of Lexpr list
  | Lcast   of Lexpr * Lexpr * Lexpr * Lexpr
  | Lcheck  of Lexpr * Lexpr
  | Lcond   of Lexpr * Lexpr * Lexpr
  | Lselect of Lexpr * (Lexpr * Lexpr) list * Lexpr
  | Lcatch  of string list * Lexpr
  | Let     of Lexpr * Ldecl
  | Lseq    of Lexpr list
```

The semantics are defined in the *denotational semantics* style.[2] A denotational semantics is based on the idea that each entity of the language that is to be described denotes a mathematical entity, e.g., numeric values

[1] This is the "abstract" syntax, given in the form of an SML data type declaration. Concrete examples will be in a hybrid syntax consisting of ALDiSP-, abstract LF-, and "semantic-level" expressions. The actual data structures employed in the compiler also contain source file positioning information; this detail has been omitted from most of the following material.

[2] A detailed introduction into denotational semantics of programming languages is given by Schmidt [106].

denote numbers, and procedures denote continuous functions on domains. A special trick (introduction of bottom elements) is used to make the functions defined in the face of non-termination. In practice, many denotational definitions look like functional interpreters, and can be directly implemented as such in any functional programming language. One of the best-known denotational definitions is given in the Scheme report; the authors felt confident in its correctness since "the semantics in this section was translated by machine from an executable version of the semantics written in Scheme itself" [27].

The ac semantics does not fully conform to "proper" denotational semantics style: it contains some "global data structures" and models LF functions as explicit "inert" closures instead of semantics-level functions.

Many languages for which denotational semantics are written are *statically typed*[3] and have two semantics: the "static" or "type" semantics defines the types of expressions, while the "dynamic" or "run-time" semantics defines the execution behaviour. The dynamic semantics is only defined on well-typed programs, i.e. programs for which the static semantics has found correct types. An example for such a language is SML [87, 88].

Since there are no static typing rules for ALDiSP, only an execution semantics exists. This semantics contains explicit and implicit tests for well-typedness.

To keep the semantics legible, it is written in the style of an SML program. It is not directly executable, since it sometimes employs "non-constructive" notation such as quantors, and meta-syntax such as "...".

In the first section, the domains used by the semantics will be presented. In the following sections, evaluation functions are defined for all syntactic forms. In specific cases, there will be discussions concerning some of the more difficult points like automatic dereferencing, blocking suspensions and automatic mapping.

The semantics was developed in parallel with an interpreter; thus they are both "tested". In form and contents, the current semantics are a major revision of the semantics published in [45], which was done prior to the interpreter implementation. The most important changes are:

- The *Result* type has been simplified; the earlier version had four kinds of results (simple results, multiple-value results, exceptions, and mismatches), the current one only two (results and exceptions).
- Tuples have been introduced, to remove the need for multiple-value results.
- Generalized arrays have replaced the vectors and matrices.
- The representation of the semantic equations has been converted from the usual "mathematical" style to the SML-like style.

3.1 Semantic Domains

In denotational semantics, the elements of computation (what we usually call "values") do not form a simple set, but rather a *domain*. A domain is a set with an internal structure given by a partial *ordering relation* and a *least element* $\perp$. Together, they define a *complete partial order (CPO)*. The domain of "all ALDiSP values" is called *Obj*. It is a *flat* domain, i.e. all "data" elements are uncomparable under the ordering relation, and "bigger" than the bottom element:

$\forall\ x \in\ Obj:\ x \sqsupseteq\ \perp$

[3] In a statically typed language, each expression has a unique type that can be computed without actually running the program.

3.1.1 Conventions and Notation

A *homogenous list* type using SML notation is assumed; the type "list of x" is notated as "x `list`". List literals are written as `[`$x_1, \ldots, x_n$`]`; the n-th element of a list x is denoted as `nth(`x`,`n`)`; the length of x is $|x|$. A list can be constructed element-wise using the `::` list-constructor, i.e. `[1,2,3]` can be written as `1::2::3::[]`.

Environment substitution is written as $Env[symbol/value]$, environment lookup is $Env[symbol]$. Looking up undefined names returns $\perp$.

"Unimportant" semantic domains and primitive functions – those that play no rôle in the evaluation process – are not described; e.g., the existence of floating point numbers is mentioned, but not their semantics.

For many purposes, unique tags are needed. These are drawn from a set *Tags* that remains unspecified; the only operation defined on tags is comparison for equality.[4]

Whenever an error situation occurs, an *error* is "returned". An erroneous program will deliver an undefined result; the `error` pseudo-function generates $\perp$. Sometimes, the error is attributed with a cause, e.g. `error("mismatch")`; this is for documentation purposes only.

Variable naming conventions are that E represents environments, S states, and o or obj objects. If some value is incrementally changed, this is usually denoted by "ticking" the successive values, e.g., $S, S', S'', \ldots$. The symbol K is used for continuation functions.

If a set consists of one constructor only, the constructor will have the same name of the set, but in lowercase.

Conditional definitions have the form

```
if ... then ...
[] ... then ...
⋮
[] ... then ...
else ...
```

The `case` conditional destructures values:

```
case expr
  of pattern1 => expr1
  [] pattern2 => expr2
  ⋮
else exprelse
```

The lambda symbol (λ) is used in modeling functions. The expression

$$\lambda\ x\ y\ z\ .\ expr$$

denotes a function of three arguments, that, when applied, evaluates by substituting the arguments in the expression. That is, the λ operator denotes anonymous functions much like the `lambda` special form in Scheme or the `fn` of SML. Because the λ-operator is lexically scoped, parameter names do not matter, i.e.

$$\lambda\ x.x + 1 \ \equiv\ \lambda\ y.y + 1$$

[4] For practical reasons, tags can be thought of as mapped to the integers. In chapter 6, tags are needed that are ordered by "creation date".

Usually, though, the names are chosen to have mnemonic value.

Whenever an expression like

$$\lambda\ [x_1, \ldots, x_n]\ .\ expr$$

is encountered, it is assumed that the function defined by the lambda is only defined on lists; names of the form x_i occuring in *expr* refer to the i-th element of the list, and n is the size of the list.

3.1.2 Data Domains

The basic data domain is that of *Objects*:

$$Obj = (Bools \cup Numbers \cup Arrays \cup Tuples \cup References \cup Functions \cup Types) \cup \{\bot\}$$

It consists of a number of standard and some ALDiSP-specific sets, and is made into a partially ordered domain by introducing a bottom element. The abstract semantics (cf. chapter 6) will introduce another such value, "top" ($\top$), which is "more defined" than any other object.

$$\begin{array}{l}
Bools = \{\texttt{true}, \texttt{false}\} \\
Numbers = Scalars \cup Complices \\
Complices = Scalars \times Scalars \\
Scalars = Integers \cup Floats \cup Times \cup Durations \\
Integers = FiniteIntegers \cup InfiniteIntegers \\
InfiniteIntegers = \{\texttt{0}, \texttt{1}, \texttt{-1}, \texttt{2}, \texttt{-2}, \ldots\} \\
FiniteIntegers = \{\texttt{int}(n, size) \mid n \in InfiniteIntegers, size \in [0 \ldots MAXINTWIDTH]\ \} \\
Floats = (\textit{IEEE 754 floating point numbers}) \\
Times = \{\texttt{time}(n) \mid n \in InfiniteIntegers\} \\
Durations = \{\texttt{duration}(n) \mid n \in InfiniteIntegers\}
\end{array}$$

These are the standard data domains. There are two kinds of integers, finite and infinite ones. Infinite integers are provided as a semantic base on which finite integers can be defined. A finite integer is an integer with a size. The size determines the rounding and overflow behaviour; a size of n implies a representation as an n-bit twos-complement number. Floating-point numbers can be defined analogously; details can be found in [46].

A special problem is posed by the existence of numeric literals: in an expression like `1+x`, the `1` does not have a unique type. This fact is modelled by representing it as an `InfiniteInteger`.[5]

Types *Times* and *Durations* are needed to represent points in time and durations between such points. Both are isomorphic to the positive integers; a program's execution starts at a time "0" and advances in time in time steps.[6]

$$Arrays = \{\texttt{array}([s_1, \ldots, s_{dim}], [e_0, \ldots, e_{s_1 * \ldots * s_{dim} - 1}]) \mid s_i \in [1..MAXSIZE], \forall\ i\ :\ e_i \in Obj\}$$

[5] In the compiler, infinite integers are represented as finite integers with a zero size. Thus, any operation that has a literal as one of its arguments will have the type of the non-literal arguments as result type, following the "result type = biggest argument type" convention.

[6] The time steps are supposed to be evenly spaced. This only matters when the scheduler is implemented, i.e. when a correlation between *Times* and some physical time is made. To the semantic functions, time is a black box.

Arrays may be of arbitrary dimension.[7] There is a maximum size for each dimension. Zero-width dimensions are not allowed. All elements of an array should be of the same type.[8] The semantics does not define the physical placement of elements in an array, it treats an array as a bag of values accessible via index tuples.

$$\mathit{Tuples} = \{\texttt{tuple}(tag, v_1, \ldots, v_i) \mid tag \in \mathit{Tags},\ \forall\, i:\ v_i \in \mathit{Obj}\}$$

A tuple is a record, i.e. a set of values that may be of different types, and are accessed by a "field name" (which is represented by its position, when written down as a list). All tuples sharing the same tag should have the same number of elements.

Tuples are primarily employed in modelling abstract data type constructors. As a side benefit, they are used to implement multiple-valued functions and exceptions. Later in the compiler, they are also used to explicitly represent environments.

$$\begin{aligned} \mathit{References} &= \{\texttt{ref}(tag) \mid tag \in \mathit{Tags}\} \\ \mathit{Tags} &= \mathit{unspecified};\ \mathit{source\ of\ unique\ tags} \end{aligned}$$

References are placeholders that refer to promises (pending call-by-need computations) and suspensions. Suspensions are held in the global state (more about the state in section 3.1.5). This "non-functional" behaviour is necessary because promises are cached and suspensions have side-effects, so all accesses to them have to be coordinated.

$$\begin{aligned} \mathit{Functions} &= \mathit{Primitives} \cup \mathit{Closures} \cup \mathit{Overloadeds} \\ \mathit{Primitives} &= \{\texttt{int_add}, \texttt{int_sub}, \ldots, \texttt{overload}, \ldots\} \\ \mathit{Closures} &= \{\texttt{closure}(tag_c, tag_l, env, vars, types, body) \mid tag_c, tag_l \in \mathit{Tags}; \\ &\qquad env \in \mathit{Env}; \\ &\qquad vars \in \mathit{Var\ list} \\ &\qquad types \in \mathit{Obj\ list}; \\ &\qquad body \in \mathit{Expr}\} \\ \mathit{Overloadeds} &= \{\texttt{overloaded}(xs) \mid xs \in \mathit{Closures\ list}\} \end{aligned}$$

There are three types of functions: primitives, closures, and overloaded functions. While primitives are not explicitly typed, closures are. Closures are first-order functions with a lexical scope. There is no special object type for Pascal-like "stack functions" (closures that are not exported from their lexical scope). An overloaded function is just an ordered collection of closures. Closures are used to model modules (cf. section 4.2.4), so no special module data type exists.

Closures consist of a tag_c which makes them unique, a tag_l that identifies the `Lambda` expression of which they are an instance, an expression that defines the body of the function, an environment holding the static bindings of the free variables occuring in the body, and a list of *types* that define the acceptable arguments.

$$\begin{aligned} \mathit{Type} = \{\ & \texttt{Obj},\ \texttt{Bool},\ \texttt{Tuple}(tag, types),\ \texttt{Time},\ \texttt{Duration},\ \texttt{Int}(n),\ \texttt{Float}(m, n), \texttt{String}, \\ & \texttt{Array}([size_1, \ldots, size_{dim}], type),\ \texttt{Proc}(types, type), \texttt{Closure}(tag),\ \texttt{Pred}(obj) \\ & \mid n, m, size_1, \ldots, size_{dim} \in \mathit{Int}; type \in \mathit{Type};\ types \in \mathit{Type\ list};\ obj \in \mathit{Obj};\ tag \in \mathit{Tag} \\ & \} \end{aligned}$$

Types denote sets of semantic values. There is a fixed set of primitive types, which correspond to the semantic value sets presented here. Some of the types are parameterized: `Int`(n) describes the set of all

[7] At the surface, ALDiSP supports only one- and two-dimensional arrays. The introduction of a general array concept proved to be a simplification.

[8] This requirement can lead to problems when numeric values of different types are concerned. It might not be sensible to force the user to write `#[1.0,1.5,2.0]` instead of `#[1,1.5,2]`.

(twos-complement) integers that can be encoded in n binary digits; `Tuple(`*tag,types*`)` is the set of all tuples having the tag *tag* and elements conforming to *types*, and `Closure(`n`)` is the set of all simple functions with a tag equivalent to n.

The type `Pred` is special; it introduces arbitrary predicates into the set of types. It is syntactically necessary since types such as `Tuple` or `Proc` have sub-types, which may be user-defined.

Due to the presence of `Pred`, type checking is a non-atomic operation, and type-inference is generally impossible: when two given types t_1 and t_2 contain `Pred` terms, one cannot even determine whether they are in a sub-type relationship. Sub-typing for atomic types is well-defined and reflects the usual arithmetic type hierarchy.

3.1.3 Results

The evaluation of an expression leads to a *Result*, which can be of two kinds:[9]

$$\begin{aligned} \mathit{Result} &= \mathit{Exception} \cup \mathit{SimpleResult} \\ \mathit{Exception} &= \{\texttt{exception}(\mathit{tag}, \mathit{obj}, \mathit{state}) \mid \mathit{tag} \in \mathit{Tags},\ \mathit{obj} \in \mathit{Obj},\ \mathit{state} \in \mathit{State}\} \\ \mathit{SimpleResult} &= \{\texttt{result}(\mathit{obj}, \mathit{state}) \mid \mathit{obj} \in \mathit{Obj},\ \mathit{state} \in \mathit{State}\} \end{aligned}$$

Their respective usage is:

- `exception(`*tag*, *obj*, *state*`)` is returned by aborting exceptions. It acts as a bottom value, which forces most semantic functions to abort any any further computation once an exception was generated. The exception thus "travels up the return stack" to the point of the innermost `guard` that can handle it. Exceptions are identified by their tag.[10]
- `result(`*obj*, *state*`)` represents the normal case where a single value is returned.

The internals of *State* are discussed in section 3.1.5. The syntactic domains (expressions, declarations, programs) are denoted by their implementation data-type names (*Lexpr*, *Ldecl*, *Lprog*).

3.1.4 Evaluation Functions: an Overview

The semantics of LF is presented as a set of semantic functions. These can be understood as "evaluation functions".

[9] Originally [45], there were two additional result types: `mismatch` was returned by mis-typed function applications; and `results(`*objs*`)` was returned by multiple-valued functions. The removal of these two simplified the semantics significantly.

[10] The first intermediate representation (i.e., the precursor of the Lambda Form) used a continuation-passing style. Because this made exception handling easier to formalize: the return jump of an aborting exception was a simple function call. This approach was abandoned for a number of reasons:

- The resulting code was bloated and hard to read; transformations were error-prone, since each function had to be transformed so as to accept and pass along two extra continuation parameters.
- I was not sure whether efficient code could be generated for this.
- At one place in the semantics (evaluation of promises and suspensions) it is necessary to implement a "catch-all" for exceptions, and it is not clear how to express this in a CPS-evaluator.

Lastly, the semantics are more "natural" using the present approach, though this is a subjective observation.

3.1.4.1 `eval`$_{\text{expr}}$

The main evaluation function is

$$eval_{expr} : Lexpr \times Context \rightarrow Result$$

The *Context* is composed of three parts:

$$Context = Env_{static} \times Env_{dynamic} \times State$$

There is an important convention: if a semantic equation contains a context `C'`, then references to `Env'`$_{\text{static}}$, `Env'`$_{\text{dynamic}}$, and `S'` will refer to the parts of this context. Likewise, the expression `C+S'` means that the state of context `C` is replaced by `S'`.

The static environment contains lexical bindings; the dynamic environment contains exception bindings. The difference between the two is that `Env`$_{\text{dynamic}}$ is passed on to a function when it is applied; while `Env`$_{\text{static}}$ is encapsulated in the functions' closure.

Environments are mappings from variables to objects:

$$Env = Var \rightarrow Obj$$

3.1.4.2 `apply`

Function application is the most complex part of the language; the large number of auxiliary functions that surround the *apply* reflects this. The signature of the core *apply* is

$$apply : Obj \times Obj\ list \times Context \rightarrow Result$$

3.1.4.3 `eval`$_{\text{decl}}$

The evaluation of declarations results in two new environments:

$$eval_{decl} : (Ldecl \times Context) \rightarrow (Env_{static} \times Env_{dynamic} \times Context \rightarrow Result) \rightarrow Result$$

The third argument to $eval_{decl}$ is an explicit continuation function called with the newly created environments. This is necessary so that exceptions occurring during the evaluation of the declaration can be handled correctly.

3.1.4.4 `eval`$_{\text{program}}$

At the program level, the evaluation semantics entails additional complexities, because side effects and the progression of suspensions in time have to be modelled.

While the evaluation of an expression is deterministic with regard to the current state, the execution of the program as a whole may be non-deterministic. Even when all wait times are points (i.e. not really intervals), the order of execution of "concurrent" suspensions is undefined.

The semantic function that models the scheduler exhibits this by being explicitly non-deterministic:[11]

$eval_{program} \; : \; Lprog \; \rightarrow \; State \; list$

Program evaluation consists of two phases. First, the program *installs* a set of suspensions. Simply evaluating the program as a declaration and an initial expression results in some "return value" (which is of no further interest) and a changed state containing suspensions waiting for time to pass and/or input to occur. This state is then handed over to the scheduler which controls the further execution of the program: abstract time elapses, I/O transactions are performed, suspensions are evaluated and new suspensions are installed. Each suspension's evaluation is purely functional in its context, but creates a modified state that may contain events and new suspensions.

$schedule \; : \; State \; \rightarrow \; State \; list$

This is the point where non-determinism occurs, since pending suspensions have no "natural" order of execution that could be used to define such an ordering.

3.1.5 State

At this point, the concept of "state" should be clarified: Conceptually, "state" describes

- the current time (*Clock*)
- the two sets of non-evaluable and waiting suspensions (*Ususps* and *Wsusps*)
- the set of unevaluated promises (*Uproms*)
- the set of evaluated references (*EvRefs*)
- the set of I/O events (*Events*)

The previous semantics ([45]) further contained an "I/O state" component. The actual implementation has shown that no such thing is needed; instead, I/O can be modelled via "faked" suspensions called "events": a writing I/O primitive will create an event that contains the information what and where to write; and the scheduler will execute the output operation at some point of time. Likewise, input operations will create suspensions that are transformed into `EvRef`s by the scheduler when data arrives. Events can be treated like opaque `Ususps` with a time frame of zero: only the scheduler knows when an event might be performed; if it is performed, it is transformed into an `EvRef` that holds some unspecified value.

$$
\begin{array}{lll}
State & = & RefDefs \times Clock \\
RefDefs & = & tag \rightarrow (Ususps \cup Blocks \cup Wsusps \cup EvRefs \cup Uproms \cup Events) \\
Ususps & = & \{\texttt{Ususp}(cond, expr, t_1, t_2) \mid cond,\ expr \in Closures; \\
 & & \qquad t_1, t_2 \in Durations\} \\
Wsusps & = & \{\texttt{Wsusp}(expr, t_1, t_2) \mid expr \in Closures; \\
 & & \qquad t_1, t_2 \in Durations\} \\
Blocks & = & \{\texttt{Block}(tag_{wait}, f) \mid tag_{wait} \in Tags,\ f : State \rightarrow Result\} \\
EvRefs & = & \{\texttt{EvRef}(obj) \mid obj \in Obj\} \\
Uproms & = & \{\texttt{Uprom}(obj) \mid obj \in Closures\} \\
Events & = & \{\texttt{Event}(u) \mid u \in implementation\ specific\ event\ description\} \\
Clock & = & Times
\end{array}
$$

[11] In the semantics of [45], the semantic function returns a set of possible *state sequences*. The current semantics does not do so since the use of explicit non-determinism makes the semantics shorter and easier to read. Besides, the all-state approach is practically non-implementable.

Ususp terms represent unevaluated suspensions; *Wsusps* are suspensions with an condition that is `true`, and *EvRefs* represent evaluated suspensions or promises.[12] *Uproms* are unevaluated promises.

The *Blocks* are special-cased *Ususps*: whenever the interpreter blocks, it generates a `Block` reference definition that contains the blocked interpreter as a function from *State* to *Result*. The second parameter is the tag that caused the block. The blocked evaluation can continue when the suspension associated with the blocking tag has been evaluated.

The *State* can be accessed like an environment: $S[t]$ gets the definition associated with the tag t; $S[t/v]$ introduces or redefines a definition.

3.2 Auxiliary Semantic Functions

This section lists some auxiliary functions that will be employed all over the rest of the semantics.

3.2.1 $\texttt{eval}_{\text{exprs}}$

$\texttt{eval}_{\text{exprs}}$ evaluates a list of expressions, carrying along the context `C` and a *continuation function* `K`. `K` is called with the resulting object list and context when the last expression of the list has been evaluated.[13]

```
eval_exprs([],C) K = K([],C)
eval_exprs(expr::exprs,C) K =
    case eval_expr(expr,C)
      of exception(x,o,S') => exception(x,o,S')
      [] result(obj,S') => if obj = ⊥
                              then result(⊥,S')
                           else
                              eval_exprs(exprs,C+S')
                                  λ(objs,C'). K(obj::objs,C')
```

$\texttt{eval}_{\text{exprs}}$ takes care of exceptions and $\perp$ values: if an exception or a bottom is encountered, it is propagated upward[14]; in this case, the continuation function is never called. In the case of $\perp$, this is necessary to guarantee the strict behaviour of the language.

3.2.2 $\texttt{deref}_{\text{objs}}$

$\texttt{deref}_{\text{objs}}$ takes an object list and delivers an object list of equal length that is guaranteed not to contain references. Since $\texttt{deref}_{\text{objs}}$ can block, it calls a continuation argument with the result instead of simply returning it. The parameter `C` holds the context (remember that the state `S` is defined implicitly as a component of the context `C`, so that `S[tag]` refers to the value of `tag` in the context `C`).

```
deref_objs([],C) K = K([],C)
deref_objs(obj::objs,C) K =
   case obj
```

[12] In the first drafts of the semantics, there were no *Wsusps*, but that implied that the schedule had to traverse all *Ususps* to look for the evaluable subset. Also, there is the possibility of a once true expression becoming false again during the waiting time; without an extra "waiting pool" it would have been impossible to correctly execute such suspensions. Of course, "correct" here means: intuitively, without there being any prior formal definition.

[13] For those readers who have no experience with CPS-semantics, it should be noted that the λ-expression denotes (or creates) the continuation function. It corresponds to an explicit reference to the "stack position" at which the "local variable" `obj` is stored.

[14] I.e., the evaluation aborts and returns the exception value.

```
    of ref(tag) =>
          case S[tag]
            of EvRef(x) => deref_objs(x::objs,C) K
            [] Uprom(x) =>
                   case eval_thunk(x,S)
                     of (x,S') =>
                       deref_objs(x::objs,C+S[tag/EvRef(x)]) K
          else
               block(tag,C)
                   λ(S). deref_objs(obj::objs,C+S) K
  else
     deref_objs(objs,C)
        λ(objs',C'). K(obj::objs',C')
```

When $\texttt{deref}_{\text{objs}}$ dereferences an **EvRef**, it calls itself with the dereferenced value. This loop is necessary to correctly cope with nested suspensions or delays.

When $\texttt{deref}_{\text{objs}}$ encounters an unevaluated promise (**Uprom**), its value is computed by $\texttt{eval}_{\text{thunk}}$. The state is updated, i.e. the **Uprom** is replaced by an **EvRef** containing the result value.

When $\texttt{deref}_{\text{objs}}$ encounters a reference that points to a unevaluated suspension, it blocks the current process by calling **block** with the current continuation. The blocked continuation captures the current context and uses it to provide the lexical and dynamical environments needed to continue the evaluation.

3.2.3 block

block generates a **Block** definition that encapsulates the continuation **K**, installs it in the state, and returns a reference to the block.

```
block(tag_wait,C) K =
    result(ref(tag_new),
           S[tag_new / Block(tag_wait,K)])
```

3.2.4 strict, $\texttt{strict}_{\text{exprs}}$

A variation of $\texttt{eval}_{\text{exprs}}$ is $\texttt{strict}_{\text{exprs}}$, which extends the functionality of $\texttt{eval}_{\text{exprs}}$ by asserting that none of the resultant objects is a reference. It dereferences all **EvRef**s, forces all **Uprom**s, and blocks when there are suspended arguments.

```
strict_exprs(exprs,C) K =
    eval_exprs(exprs,C)
        λ(objs,C').
            deref_objs(objs,C') K
```

The simple **strict** takes a result, a context and a continuation; it calls the continuation with the unpacked and dereferenced object and state.

```
strict res C K =
  case res of
    result(x,S') => deref_objs([x],C+S') (λ([x'],C'') . K(x',C''))
  else res
```

3.2.5 $\texttt{eval}_{\text{thunk}}$, $\texttt{norm}_{\text{result}}$

A *thunk* is a parameterless function. Thunks are employed to "freeze" the evaluation of expressions that are suspended or delayed.

```
eval_thunk(x,S) =
    norm_result(deref_objs([x],(∅,∅,S)) λ (x',C'). apply(x',C'))

norm_result(res)
    case res
      of result(x,S') => (x,S')
      [] exception(x,o,S') => (⊥,S')
```

A thunk is expected to return a `result`, therefore all `exception` results are caught and replaced by ⊥. The function $\texttt{norm}_{\text{result}}$ performs this "normalization"; it transforms a result into an object and a state. The tuple `(∅,∅,S)` is a context that contains a state, but no variable bindings.

3.3 Evaluating Expressions

3.3.1 Lambda: Closure Creation

A `Lambda` expression creates a closure with a typed argument list. The closure represents a function that, when applied to an argument list, confirms that each arguement matches its corresponding type, and evaluates the function body within the current lexical environment updated with the argument bindings.

The transformation phase creates `Lambda` expressions when it encounters ALDiSP function definitions (cf. section 4.2.2), module definitions (cf. section 4.2.4) and abstract data type definitions (cf. section 4.2.6.2). Furthermore, they are used to implement the thunks in `delay` and `suspend` expressions (cf. sections 4.4.4 and 4.4.5). The semantics itself creates new `Lambda`s on the fly as expands auto-mapped applications to nested standard applications (cf. section 3.6).

```
eval_expr(Lambda(tag_lambda,[(v_1,p_1),...,(v_n,p_n)],expr),C) =
    eval_exprs([p_1,...,p_n],C)
        λ([t_1,...,t_n],C').
            result(closure(tag_closure,tag_lambda,E_static(C)∩ freevars(expr),
                           [v_1,...,v_n],[t_1,...,t_n],expr),S')
```

$\texttt{tag}_{\text{closure}}$ is a freshly allocated tag that uniquely identifies each closure generated at run-time.

`freevars(`*expr*`)` denotes the set of syntactically free variables in `expr`. The static environment which is encapsulated in the closure is restricted to this set; it is not necessary to put bindings of variables into the closure that do not occur in the code that it closes over.

The argument types $p_1, \ldots, p_n$ are evaluated at closure creation time; the resulting type objects $t_1, \ldots, t_n$ are made part of the closure.

3.3.2 Lvar: Variable Look-Up

Evaluation of variables is performed by looking them up in the static or dynamic environments. These environemnts are part of the evaluation context.

```
eval_expr(Lvar(var),C) =
    case E_static(C)(var)
```

```
    of ⊥ => case E_dynamic(C)(var)
                 of ⊥ => error("undefined variable")
                 [] x => result(x,S)
    [] x => result(x,S)
```

Only if a variable name is not defined in the lexical environment, it is looked for in the exception environment. If it is not found there either, this is an error. Some of these errors can not be detected statically; the implemented transformation phase looks for statically unbound variables that are never bound dynamically; this check can cheaply be done as part of the α-conversion (cf. section 4.5.5). It cannot be statically guaranteed that dynamically bound variables are present in the environment.[15]

3.3.3 `Lit`*: Literal Values*

```
eval_expr(Lit(x),C) = result(inject_obj x,S)
```

The `Lit` construct directly represents a semantic value.[16] A value has to be "injected" into the `Obj` domain; this is done by the (otherwise unspecified) $\texttt{inject}_{\text{obj}}$ function. While this function looks like a mere techniciality in the semantics, it becomes a real operation in the actual abstract interpreter, where the domain of abstract values may differ greatly from the domains of the semantics.

3.3.4 `Lapp`*: Function Application*

To evaluate a function application, the function and all its arguments must first be evaluated in some order. This order is rather arbitrary in principle; here it specified as left-to-right, but any sequential order would suffice.[17] Then the application itself is performed.[18]

$\texttt{apply}_{\text{mappable}}$ handles all details of application – it confirms that the function can be applied at all to these arguments, and calls the "real" `apply`; if there is an argument type mismatch, auto-mapping is tried.

```
eval_expr(Lapp(exprs),C) =
if |exprs| = 0
    then error("empty application")
    else eval_exprs(exprs,C)
         λ(obj_func::obj_args,C').
             deref_objs([obj_func],C')
                 λ([obj_func'],C'').
```

[15] This can be shown by example:

```
func needs_f(b) = f(b)
func might_crash(x) =
  if complex_predicate(x) then needs_f(x)
  else guard needs_f(x)
         on f(a) = a
       end
  end
```

If `f` is not defined in the lexical context or in the dynamic environment which is active when `might_crash` is called, a semantic error will occur if `complex_predicate` is `false` for `x`.

[16] Such constructs are often called "constants", which is totally wrong. A *constant* is an expression guaranteed to have the same value in all evaluation contexts; a *literal* is the "textual" representation of a value. Of course, all literals are constants (except in old FORTRANs); but not all constants need be literals! Most programming languages allow only literals as constants; but that should not obscure the terminology.

[17] It is not only bad practice to rely on order of evaluation in function application; such practises also make parallelization nearly impossible. It would be a nice frill if the compiler could find any such dependencies.

[18] Macros can be incorporated as an evaluation mechanism by first evaluating the functional position, and, if it turns out to be a macro, substituting the expressions and evaluating the result. The reason why this strategy was not adopted is elaborated in section 3.8.1.

```
applymappable(objfunc',objargs,C'')
```

The object that represents the function is piped through $\texttt{deref}_{objs}$ to ensure that it is not a reference. If the other objects were also dereferenced prior to application, the execution behaviour would change drastically: it would be impossible to pass pending suspensions to functions.[19]

3.3.5 `Lcheck`: *Asserting Types*

`Lcast` and `Lcheck` are the LF equivalents of ALDiSP type-casts and type-checks.

In ALDiSP, a *type check* is intended to make an assertion: `[`*type*`]`*expr* asserts that *expr* is of type *type*. If this assertion is violated, the further execution of the program is undefined. An interpreter should convert all `Lcheck`s into tests; a compiler can assume that the type given in a `Lcheck` is correct.

If *expr* denotes a "data" value, checking amounts to simple function application: Since types are interchangeable with the predicates that define them, the *type* can be applied to the *expr*; if the result is `true`, everything is fine, otherwise there is an error.

If *expr* denotes a *function* and *type* a function type, the situation is different. It is in general impossible to assert that a given function `f` has some requested type. For a number of reasons (predicates-as-types, overloading, etc.), ALDiSP does not support type inference. It is therefore impossible to implement an `Lcheck` as a compile-time verification.[20]

The only *safe* way of implementing type checks on functions works as follows: an expression `[a -> b]f` (Casting "`f`" to the type "function from `a` to `b`") is transformed to `lambda(tmp:a).([b](f tmp))`: A new function of argument type "`a`" is constructed that, when called, executes "`f`" and verifies that the result is of type "`b`". By using this transformation strategy, the argument and result type checks are deferred to the function's application time. Furthermore, the domain of the function is now guaranteed not to exceed the *type* specification.[21]

```
evalexpr(Lcheck(exprt,expre),C) =
    evalexprs([exprt,expre],C)
        λ([objt,obje],C').
             derefobjs([objt],C')
                 λ([objt'],C'').
                     case objt'
                       of Proc([a1,...,ai],r) => checkproc([a1,...,ai],r,obje,C'')
                       [] Obj => result(obje,S'')
                     else
                        strict apply(objt',[obje],C'') C''
                               λ(x,C''').
                                   if x = true
                                     then
                                       result(obje,S''')
                                   else
                                     error("type mismatch")

checkproc([a1,...,ai],r,obje,C) =
    result(closure(tagclosure/new,taglambda/new,∅,[v1,...,vi],[a1,...,ai],
```

[19] Dereferenced application *is* a possible optimization, comparable to call-by-value in a lazy language. The strictness analysis outlined in chapter 4.5.9 could be used to generate annotations to guide such a behaviour.

[20] Most `Lchecks` are nevertheless eliminated at compile time, since manys types are verified by the abstract interpreter, and the partial evaluator removes the matching `Lcheck`s as dead code.

[21] This is important, since a type check can change the behaviour of a checked value, if that value is a function that is later overloaded, and the checked type denotes a function of reduced domain.

```
                    Lcheck(Lit(objr),Lapp([Lit(obje),Lvar(v1),...,Lvar(vi)])))
        ,S)
```

The type has to be dereferenced, since *its* type must be known. The obj to test is *not* dereferenced, since this would destroy the semantics of

```
[Obj] suspend ... end
```

This should *not* block, since any possible result is an object.

3.3.6 Lcast*: Casting Types*

The `Lcast` form tries to *force* (or *coerce*) an *expr* to a type *type*. For example, the ALDiSP expression `[!Real]17`, which translates to an `Lcast`, should result in the floating point number `17.0`. In other words, the `Lcast` form is a general way to specify an "appropriate" conversion function.

The two pre-defined variables, `coerce` and `coerce_base` implement the underlying functionality. Both can be redefined by the user. They are given to the `Lcast` form as third and fourth parameters. This explicit treatment is necessary since the transformation phase reorders all declarations in the program; if no explicit connection between the "magic" cast function variables and the `Lcast` forms would be made, a declaration involving an `Lcast` could be moved to a position where the casting function is not yet defined.[22]

While `coerce` has to be bound to an overloaded function that takes one argument (the object to coerce), `coerce_base` should be bound to a function of two arguments (the object and the goal type).

If `Lcast` is called with a basic (non-user-defined) type as goal, `coerce_base` is called with the obj_{type} and obj_{expr} as arguments. The standard environment has to provide appropriate functions to convert between base types. Since the set of base types is implementation-dependent (to provide for application-specific numeric or I/O types), the library writer may need this extra flexibility.

If `Lcast` is called with a predicate as goal type, the overloaded `coerce` function is dismantled into its closures, and each of those in turn is applied to the $\texttt{obj}_e$. The first such application that delivers a result that tests `true` under the type is returned as the value of the `Lcast`.[23] The applications are ordered according to the overload specification; i.e. the most recently defined coerce functions are tried first.

```
evalexpr(Lcast(exprt,expre,ecf,ecbf),C)=
    evalexprs([exprt,expre,ecf,ecbf],C)
        λ([objt,obje,objcf,objcbf],C').
             derefobjs([objt,objcf,objcbf],C')
                 λ([objt',objcf',objcbf'],C'').
                     if objt' = Obj then result(obje,S'')
                     [] objt' ∈ TypesBasic then castbase(objt',obje,objcbf',C'')
                     else castgeneral(objt',obje,objcf',C'')

castbase(objt,obje,objcbf,C) =
     apply(objcbf,[objt,obje],C)

castgeneral(objt,obje,objcf,C) =
    case objcf
      of overloaded(c1,...,cn) =>
           let
```

[22] Exactly this was the problem with an earlier approach, in which `coerce` and `coerce_base` were "magic" variable names that were picked out of the static environment by the semantic function. A first try to "repair" the semantics by moving these variables into the dynamic environment worked, in that it protected them from unwanted α-conversion and other transformation rules. The final, current model of a four-parameter cast simplifies things a lot.

[23] This rather inefficient mechanism is the only way to cope with the existence of arbitrary predicate types.

```
        ∀ i ∈ 1..n:
            (try_i,S_i) = norm(apply(c_i,[obj_e],C))
            test_i = check(obj_t,try_i,C+S_i)(λ(ok,C').ok)
    in
        if ∃ k: test_k = (true,s) ∧ ∀ 0 < j < k:test_j ≠ (true,s)
            then try_i
        else
            error("no cast found")
```

The $\texttt{cast}_{general}$ function creates two families of indexed temporaries, namely $\texttt{try}_i$ and $\texttt{test}_i$. Their order of execution is not important, since each application is done in a "clean" context, and only the return state of the result that is finally chosen is returned.[24]

The `check` auxiliary function applies a predicate to an object and returns a boolean value. It is defined in section 3.5.1.

3.3.7 Lcond*: Two-Way Conditional*

`Lcond` models the standard `if` operator. It is *not* auto-mapped, therefore the ALDiSP expression

`if #[true,false] then 1 else 2 end`

is erroneous, and does *not* evaluate to `#[1,2]`. The first expression (i.e., the condition) must be available as a basic value and is therefore passed through $\texttt{strict}_{exprs}$. This will block the whole `Lcond` if e_1 is a pending suspension.

```
eval_expr(Lcond(e_1,e_2,e_3),C) =
    strict_exprs([e_1],C)
        λ([o_1],C').
            case o_1
              of true => eval_expr(e_2,C')
              [] false => eval_expr(e_3,C')
            else error("non-boolean argument")
```

3.3.8 Lselect*: N-Way Conditional*

`Lselect` operates in a similar manner to the `switch` of `C` or the `case` in Scheme: a selection value (e_{sel}) is evaluated, the result is taken as index into a list of expressions, each of which is tagged by a key value. The first "matching case" is chosen for evaluation; if no case matches, the default $e_{default}$ is taken. `Lselect` is strict in the selection value.

```
eval_expr(Lselect(e_sel,[(c_1,e_1),...,(c_n,e_n)],e_default),C) =
    strict_exprs([e_sel,c_1,...,c_n],C)
        λ([o_sel,o_1,...,o_n],C').
             if ∃ i: o_sel = o_i ∧ ∀ j<i: o_sel ≠ o_j
                 then
                     eval_expr(e_i,C')
             else
                 eval_expr(e_default,C')
```

[24]This "forgetting" of state changes can be very hard to implement in an interpreter that maintains the state via side effects.

3.3.9 `Lseq`*: Sequences of Expressions*

`Lseq`-expressions model expression sequences with automatic synchronization: Expressions are evaluated from left to right; the value of the last expression is the value of the sequence. If an expression returns (a reference to) an unevaluated suspension, the execution of the rest sequence blocks.

```
eval_expr(Lseq(exprs),C) =
    case exprs
      of []     => error("empty sequence")
       | [expr] => eval_expr(expr,C)
       | expr::exprs =>
           strict_exprs([expr],C)
             λ([obj'],C').
                   eval_expr(Lseq([expr_2,...,expr_n]),C')
```

In order to avoid an unnecessary auxiliary function, the rest of the sequence is packed up in a new `Lseq` and evaluated.

3.3.10 `Lcatch`*: Handling Exceptions*

An `Lcatch` catches exceptions. The `Lcatch` contains one sub-expression and a list of exception tags. The sub-expression is evaluated; if it generates an exception that is listed amon¡gst the tags, the exception is unpacked and its value is returned as an ordinary result. Otherwise, the value (be it result or unmatched exception) is returned unchanged.

```
eval_expr(Lcatch([tag_1,...,tag_n],expr),C) =
    case eval_expr(expr,C)
      of result(x,S') => result(x,S')
      [] exception(tag,obj,S') =>
            if tag ∈ {tag_1,...,tag_n}
              then
                result(obj,S')
              else
                exception(tag,obj,S')
```

3.3.11 `Let`*: Local Definitions*

The evaluation of local definitions is modelled by the evaluation rules for declarations, $\mathtt{eval}_{decl}$. $\mathtt{eval}_{decl}$ evaluates a definition, which might consist of sub-definitions, and calls its continuation with two sets of bindings (one for $\mathtt{Env}_{static}$, one for $\mathtt{Env}_{dynamic}$) and a new context. That context contains state changes only; the environments are merged with the root environment when the `expr` is applied.

Like $\mathtt{eval}_{exprs}$, $\mathtt{eval}_{decl}$ takes a continuation argument in order to be able to handle exceptions: when one of the declarations raises an exception, evaluation aborts.

```
eval_expr(Let(expr,decl),C) =
    eval_decl(decl,C)
        λ(E_s,E_d,C').
             eval_expr(expr,C'+E_s+E_d)
```

3.4 *Evaluating Declarations*

3.4.1 Ldecl: *The Simple Declarations*

An *Ldecl* is the "atomic" declaration. An expression is evaluated; the result is bound to a name. If the expression evaluates to an exception or ⊥, further evaluation is aborted; otherwise, the continuation function K is called with the generated environments and the updated context. The boolean parameter determines whether the generated binding shall be part of the static or the dynamic environment.

```
eval_decl(Ldecl(var,static,expr),C) K =
    eval_exprs([expr],C)
        λ([obj],C').
           let
             env = λ x . if x = var then obj else ⊥
           in
             if static
               then
                  K(env,∅,C')
               else
                  K(∅,env,C')
```

3.4.2 Lpard: *Parallel Declarations*

A `Lpard` is a declaration consisting of a list of declarations to be evaluated in parallel. "In parallel" means that the definitions are evaluated in the same environment; the declarations are ordered in time so that side effects can accumulate.

```
eval_decl(Lpard(decls),C) K =
    eval_pard(decls,C,∅,∅) K

eval_pard([],C,E_s,E_d) K = K(E_s,E_d,C)
eval_pard(decl::decls,C,E_s,E_d) K =
    eval_decl(decl,C)
        λ(E_s',E_d',C').
            eval_pard(decls,C',E_s+E_s',E_d+E_d') K
```

The `+` indicates that the environments are added, not replaced. No order is implied; a parallel definition

```
par
  x = 4
  x = 5
end
```

creates a lexical environment with an undefined value of `x` (either 4 or 5).[25]

3.4.3 Lseqd: *Sequential Declarations*

A `Lseqd` is a declaration consisting of a list of declarations to be evaluated sequentially, i.e. each declaration can access the variables defined in the preceding declarations.

```
eval_decl(Lseqd(decls),C) K =
    eval_seqd(decls,C,∅,∅) K
```

[25] Such situations cannot occur in ac, since ALDiSP programs contain only sequential definitions, and the transformation rules would not parallelize definitions of the same variable.

```
evalseqd([],C,Es,Ed) K = K(Es,Ed,C)
evalseqd(decl::decls,C,Es,Ed) K =
    evaldecl(decl,C+Es+Ed)
        λ(C').
            evalseqd(decls,C + S') K
```

One small problem is the accumulation of environments in the context: In a sequence of declarations, the visibility of a new declaration starts with the next declaration in the sequence. This is modelled by adding the new declarations to the context before evaluating each new declaration. However, the continuation must be called with the unchanged context, so the added bindings have to be removed before continuing with the rest sequence. This is modelled by the recursive call with `C + S'`, i.e. only the changed state is carried over.

3.4.4 `Lfixd`: *Recursive Declarations*

A `Lfixd` is a *recursive* (fixpoint) declaration. The sub-declarations are evaluated in their own visibility scope:

```
evaldecl(Lfixd(decls),C) K =
    let
      (C',Es',Ed') = fix λ(C'',Es,Ed).
                                            evalpard(decls,C'',Es,Ed)
                                                λ(C''',Es',Ed').
                                                    (C''',Es',Ed')
    in
      K(C',Es',Ed')
```

The new environment is defined to be the fixpoint under the parallel declaration, i.e. the environment that does not change when the bindings of the declaration are added. This implies that it already contains said bindings, and is therefore recursive.

The `fix` operator can either be taken as primitive, or defined as an iterative approximation process:[26]

```
fix f =
  let
     approx(x) =
         let
            x' = f x
         in
            if x' = x  then x else approx(x')
  in
     approx(⊥)
```

The actual interpreter may employ a different method (cf. section 6.5.3).

3.5 *Application Rules*

The main bulk of the *apply* function comprises the auto-mapping mechanism treated later. The core *apply* is a dispatch rule that divides the domain of *apply* into manageable portions, each of which has quite distinct behaviour.

```
apply(func,args,C) =
   if func ∈ Closures    then applyclosure(func,args,C)
```

[26]Formally, `fix` computes the fixpoint of a function of one argument. To apply it to a function of n arguments, these can be treated as tuples, with $\perp \sqsubset$ `<x1,...,xn>` .

```
   [] func ∈ Overloadeds then apply_overloaded(func,args,C)
   [] func ∈ Primitives  then apply_primitive(func,args,C)
   [] func ∈ Arrays      then apply_array(func,args,C)
   [] func ∈ Type        then apply_type(func,args,C)
   else error("is no function")
```

It is important to note that the core apply is *unchecked*: The test for applicability is isolated in an extra function, $\texttt{test}_{apply}$:

```
test_apply(func,args,C) K =
   if func ∈ Closures then test_apply/closure(func,args,C) K
   [] func ∈ Overloadeds then test_apply/overloaded(func,args,C) K
   [] func ∈ Primitives then test_apply/primitive(func,args,C) K
   [] func ∈ Arrays then test_apply/array(func,args,C) K
   [] func ∈ Type then test_apply/type(func,args,C) K
```

The `test` functions cannot generate errors, they are total. They are isolated from the `apply` because the automapping mechanism has to search for mapping possibilities by speculatively disassembling data structures and applying the results to functions. By defining an extra test function tree, tests can be done without causing unnecessary errors. In the following, `test` functions are presented next to the `apply` functions they guard.

3.5.1 $\texttt{apply}_{closure}$

A closure is applied by adding the argument bindings to the static environment that is part of the closure, and evaluating the body of the closure in the updated environment.

```
apply_closure(closure(tag_closure,tag_lambda,env,[v_1,...,v_n],[t_1,...,t_n],expr),
              [a_1,...,a_n],C) =
    eval_expr(expr,C[E_static/(env[v_1 / a_1,...,v_n / a_n])])
```

A closure can be applied to an argument list if all arguments match:

```
test_apply/closure(closure(tag_closure,tag_lambda,E,[v_1,...,v_n],[t_1,...,t_n],expr),
                   [a_1,...,a_k],C) K =
    if n = k
      then
        check_list [(t_1,a_1),...,(t_n,a_n)] C K
      else
        K(false,C)

check_list tests C K =
    case tests
      of [] => K(true,C)
       | (t,a) :: rest => check(t,a,C)
                              λ (ok,C') .
                                 if ok
                                    then
                                       check_list rest C' K
                                    else
                                       K(false,C')

check(obj_type,obj_arg,C) K =
  deref_objs([obj_type],C)
    λ([obj_type'],C').
```

```
    strict apply(obj_type,[obj_arg],C') C'
        λ(res,C'').
            case res
              of true => K(true,C'')
               | false => K(false,C'')
            else (warning("is no type"); true)
```

The `check` function will be used again in the auto-mapping section. If one of the parameter types cannot be applied to its arguments, a warning should be given. It might as well be an error. This is one of the situations where an error is signalled much too late; it should have been given when the closure was created.

The call to `apply` poses a small problem, since even this simple ALDiSP function drives the interpreter into a loop:

```
func semanticKiller(x:semanticKiller) = 42
```

Another, very similar problem stems from the recursive dereferencing:

```
rec
  x = delay(x)
end
```

In the current implementation of `ac`, these problems are considered so absurd that no provisions are made to catch them as errors.

3.5.2 $\texttt{apply}_{\text{overloaded}}$

An overloaded function is applied to the first member closure that matches the arguments.

```
apply_overloaded(overloaded([c_1,...,c_k]),args,C) =
    match_overload(overloaded([c_1,...,c_k]),args,C)
        λ(index,C').
             apply_closure(c_index,args,C')
```

$\texttt{match}_{\text{overload}}$ finds the first matching closure and returns its index, or 0 if there is none. The latter behaviour is needed for the test function; when $\texttt{apply}_{\text{overload}}$ is called, a preceding $\texttt{test}_{\text{overload}}$ has verified that a valid index (> 0) exists.

```
match_overload(overloaded([c_1,...,c_k]),args,C) K =
    find_first([c_1,...,c_k],1,C) K

find_first(closures,index,args,C) K =
    case closures
      of [] => K(0,C)
       | c::cs => test_apply/closure(c,args,C)
                      λ(ok,C') .
                          if ok
                            then
                              K(index,C')
                          else
                            find_first(cs,index+1,args,C')

test_apply/overload(func,args,C) K =
    match_overload(func,args,C)
        λ(index,C') => K(index>0,C')
```

3.5.3 apply$_{\text{primitive}}$

There are two kinds of primitives: those that are internal to the interpreter and may access and change the state; and those that perform side-effect free operations on semantic values. The latter ones share the common property that they are *strict*, i.e. access the values of all their arguments. They are of no further interest to this semantics, since strict primitives are "atomic" and cannot raise exceptions or otherwise disturb the flow of control.

There are only a few non-strict primitives, namely those needed to access and test references, to create tuples, and to create arrays.

```
apply_primitive(p,args,C) =
    if p ∈ Primitives_strict
      then
        deref_objs(args,C)
            λ(args',C').
                result(apply_primitive/strict(p,args'),S')
    else
        apply_primitive/nonstrict(p,args',C)

apply_primitive/strict(p,args) =
  case (p,args)
    of (overload,[closure(x),closure(y)]) => overloaded([closure(x),closure(y)])
    [] (overload,[closure(x),overloaded([y_1,...,y_n])])
                                          => overloaded([closure(x),y_1,...,y_n])
    [] (select,[index,tuple(tag,x_1,...,x_n]) =>
        if index ∉ Integers ∨ index<0 ∨ index>n then error()
        [] index = 0 then tag_x
        else x_index
    [] (or,[bool_1,...,bool_n]) => bool_1 ∨ ... ∨ bool_n
    [] (and,[bool_1,...,bool_n]) => bool_1 ∧ ... ∧ bool_n
    [] (tuple_type,[tag,type_1,...,type_n]) => Tuple(tag,type_1,...,type_n)
    [] (func_type,[type_1,...,type_n,type_0]) =>
         Proc([type_1,...,type_n],type_0)
    [] (is,[tuple(tag_tuple,args),closure(tag_closure,tag_lambda,...)]) =>
                        if (tag_lambda,tag_tuple) ∈ list-of-constructors
                          then true
                          else false
    [] (is,[tuple(tag_tuple,[]),tuple(tag_tuple',[])]) =>
                        if tag_tuple = tag_tuple'
                          then true
                          else false
    [] (io_op,parameters) =>
                result(ref(tag_io), S[tag_io/Event(parameters)])
                where
                   tag_io = (a new reference tag)
    [] (+,[int(n,s_n),int(m,s_m)]) => int(n+m,max(s_n,s_m))
    [] ...many more...
  else error("primitive unknown or does not match")

apply_primitive/nonstrict(p,args,C) =
    case (p,args)
      of (_deref,[ref(tag)])   => case S[tag]
                                     of EvRef(x) => result(x,S)
```

```
                                 else error()
    [] (_is_avail,[ref(tag)]) => case S[tag]
                                   of EvRef(x) => result(true,S)
                                   [] Uprom(x) => result(true,S)
                                 else result(false,S)
    [] (_is_avail,nonReferenceValue) => result(true,S)
    [] (_tuple,[tag,v1,...,vn] => result(Tuple(tag,v1,...,vn),S)
    [] (_delay,[closure(x)]) =>
                       result(ref(tag_delay),
                              S[tag_delay/Uprom(closure(x))]
                       where
                          tag_delay = (a new reference tag)
    [] (_suspend,[closure(x),closure(y,),t1,t2]) =>
                       result(ref(tag_susp),
                              S[tag_susp/Ususp(closure(x),closure(y),t1,t2)]
                       where
                          tag_susp = (a new reference tag)
    [] (_mk_exc,[tag,value]) => exception(tag,value,S)
    [] error("primitive unknown or does not match")
```

$\texttt{Primitives}_{\text{strict}}$ is the implementation-dependent set of primitive function names. For all primitives that are not listed in the semantics, behaviour is unspecified. In particular, it is not guaranteed that all primitives are total.

The `is` primitive provides type testing of constructed values. A global "list of constructors" is created by the transformation phase (cf. section 4.2.6.3). It associates each tuple tag with the lambda tag of the constructor that was used to create it.

The `io_op` primitive creates an I/O event. The arguments to `io_op` are implementation-dependent, as are the parameters stored in the `Event` reference definition.

```
test_apply/primitive(func,args,C) K = K(true,C)
```

Primitives are inherently untyped; they are not to be used directly by the programmer. It is the responsibility of the libraries to provide type-secure interfaces to the primitives. For example, the `int_add` primitive should be employed in a context like

```
func +(a:Integer,b:Integer) = `int_add`(a,b)
```

3.5.4 $\texttt{apply}_{\text{array}}$

ALDiSP has no operator or special syntax for array lookup. Instead, an array is treated like a function: when it is applied to appropriate indices, it delivers the indexed object.

```
apply_array(array([size1,...,size_dim],elements),[arg1,...,argn],C) =
    deref_objs([arg1,...,argn],C)
        λ([i1,...,i_dim,],C').
            elements(i1,...,i_dim)
```

The internal row/column structure of two- or more-dimensional arrays is not specified. The definition models an array as a bag of indexed elements that can somehow be looked up. Correct typing of the arguments (integers in the range of the array) is guaranteed by the test function:

```
test_apply/array(array([size1,...,size_dim],elements),[arg1,...,argn],C) =
    deref_objs([arg1,...,argn],C)
```

```
λ([arg₁',...,argₙ']).
    dim = n ∧ ∀ i ∈ 1..dim : argᵢ ∈ Int ∧ argᵢ' ≥ 0 ∧ argᵢ' < sizeᵢ
```

An array is applicable to an argument list if all arguments are positive integers in the range of the array.[27]

3.5.5 $\texttt{apply}_{\text{type}}$

The application of basic types is hard-wired into the semantics. One special case is the type `Obj`, which is `true` for all arguments. Due to this fact, there is no need to dereference the argument if the type is `Obj`. This special case treatment is more than a mere optimization: since all arguments to functions are typed, and an object's type is tested by applying the type to the object, default dereferencing would make it impossible to write functions that accept pending suspensions as their arguments.

```
apply_type(type,[obj],C) =
   if type = Obj then true
   [] type = Pred(t) then check(t,obj,C)
   else
     deref_objs([obj],C)
        λ([obj'],C').
           is_base(obj',type)
```

```
is_base(true,Bool) = true
is_base(false,Bool) = true
is_base(tuple(tag_x,v_1,...,v_n),Tuple(tag_y,t_1,...t_m)) =
          tag_x = tag_y ∧ n = m ∧ check(t_1,v_1,C) ∧ ... ∧ check(t_n,v_n,C)
is_base(time(x),Time) = true
is_base(int(x,s),Int(n)) = s ≤ n
is_base(float(x,sm,sn),Float(m,n)) = sm ≤ m ∧ sn ≤ n
is_base("...",String) = true
is_base(array(...),Array) = true
is_base(x,Proc(types,t)) = true
is_base(closure(tag_x,...),Closure(tag_y,...)) = tag_x = tag_y
is_base(x,Pred(obj)x) = check(obj,x,C)
```

Application of a type is handled by listing the possible cases. One type warrants special treatment: `Proc`. It is forbidden to apply function types, since they are undecidable.[28] The sole function of them is therefore to *declare* a function's type, not to check it.

```
test_apply/type(type,args,C) K =
    K(|args| = 1,C)
```

All types are applicable to all singleton argument lists.

3.6 *Auto-Mapping*

The auto-mapping mechanism is the most complex part of the semantics. The general idea behind auto-mapping is this: There exist a number of "auto-mappable homogenous collection" types, namely different kinds of lists and arrays. When a function *not* defined on these types is applied to one of them, it is mapped to each of the components. A simple example is the application `[1,2,3] + [10,20,30]`, which results in `[11,22,33]`. Though a simple paradigm, the concepts can be extended to more complex examples:

[27] The actual implementation employs slightly different means; $\texttt{apply}_{\text{array}}$ generates code for a `_lookup_array` primitive.

[28] The current implementation of `ac` emits a warning when a `Proc` is applied to an object, and approximates it safely: for function-typed arguments, `true` is returned; for non-function-typed arguments, `false` is returned. If a `Bool` were returned, the reconstruction phase would try to implement a run-time type from the `Proc`.

- If one argument is of a non-mapped type, it is "extended":
  `[1,2,3] + 100` → `[1,2,3] + [100,100,100]` → `[101,102,103]`
- Auto-mapping may be nested:
  `[[1,2],[3,4]] + [[10,20],[30,40]]` → `[[11,22],[33,44]]`
  The extension rule still applies:
  `[[1,2],[3,4]] + 100` → `[[101,102],[103,104]]`
- There is one precedence rule: Lists map before arrays. That is,
  `[1,2,3] + (10,20)` → `([1,2,3]+10,[1,2,3]+20)` → `([11,12,13],[21,22,23])`
  This precedence was chosen because lists of arrays are a more natural data type in DSP applications than arrays of lists.

There are a few restrictions: lists have to be of equal length, and arrays have to be of same rank and size, i.e., adding a three-element list to a two-element list is not permitted. Similarly, vectors cannot be added to matrices.

The auto-mapping semantics is defined via "code generation". A function application is mapped by rewriting it and evaluating the generated LF code. A few examples for the kind of generated code are:

```
[1,2,3]+[4,5,6]  ⟶  map_array(λ(v1, v2).(v1+v2), [1,2,3],[4,5,6])
[1,2,3]+100      ⟶  map_array(λ(v1).(v1+100), [1,2,3])
[1,2,3]+(4,5,6)  ⟶  map_list(λ(w1).(map_array(λ(v1)(v1+w1), [1,2,3]), [4,5,6])
```

The existence of primitive functions `map_array` and `map_list` is assumed. Their implementation is somewhat machine-dependent, since the layout of arrays is not specified in the semantics, and will depend on the target architecture.

3.6.1 $\texttt{apply}_{\text{mappable}}$

The entry point to the `apply` complex is $\texttt{apply}_{\text{mappable}}$; it is called by $\texttt{eval}_{\text{expr}}$`(Lapp(...))`.

$\texttt{apply}_{\text{mappable}}$ checks the arguments and determines whether a simple application is present or not. In the latter case, it attempts to find a way of mapping the function to its arguments.

The $\texttt{test}_{\text{apply}}$ function and all its sub-functions have to take explicit continuations, since they may force the arguments in the process of testing them, i.e. they can block.

```
apply_mappable(func,args,C) =
   test_apply(func,args,C)
      λ(ok,C').
         if ok then
           apply(func,args,C')
         else
           automap(func,args,C')
```

The `automap` function needs to inspect the `args` in order to recognize mappable types, hence it first removes any references:

```
automap(func,args,C) =
    deref_objs(args,C)
        λ(args',C').
            automap_list(args',C')
```

There are only two basic mappable types: Lists and arrays. This is reflected by the fact that there are two `automap` functions, one for lists and one for arrays. Each searches for occurrences of "its" mappable types; if it finds any, it generates a mapping function with appropriate types and tests this function for applicability. If the test succeeds, an auto-mapping is found; if it fails, a recursive call to $\texttt{apply}_{\text{mappable}}$ is tried (there may be another nesting level to un-map); if that fails, the function application is erroneous.

```
automap_list(func,[arg_1,...,arg_n],C) =
   if ∃ i ∈ 1..n : arg_i = pair(head_i,tail_i)
      ∧ ∀ i,j : i≠ j ∧ arg_i ∈ List ∧ arg_j ∈ List ⇒ |arg_i| = |arg_j|
     then
       let
          loop([]) = ([],[],[])
          loop(arg::args) =
            let
              params_inner,vars,types,params_outer = loop(args)
            in
              if arg = pair(head,tail)
                then
                  Lvar(var_tmp)::params_inner,
                  var_tmp::vars,
                  Obj::types,
                  arg::params_outer
                else
                  Lit(arg)::params_inner,
                  vars,
                  types,
                  params_outer
          params_inner,vars,types,params_outer = loop(args)
          c = closure(tag_closure/new,tag_lambda/new,∅,vars,types,Lapp(Lit(func)::inners))
       in
         apply_mappable(map_list,closure::params_outer,C)
   else
     automap_array(func,[arg_1,...,arg_n],C)
```

The transformation loop traverses the arguments and creates four lists from it: the *inner arguments*, the *outer arguments*, the *types* and the *parameters*. The outer arguments are those arguments that are to be mapped; there is one parameter for each of the outer variables. The inner arguments are made up from the original arguments, with the mapped arguments replaced by their corresponding parameters. The types are the parameter types of the generated lambda expression.

```
automap_array(func,[arg_1,...,arg_n],C) =
   if ∃ i ∈ 1..n : arg_i = array(sizes_i,elems_i)
      ∧ ∀ j ∈ 1..n: if arg_j = array(sizes_j,elems_j) ⇒ sizes_i = sizes_j
     then
       let
          loop([]) = ([],[],[])
          loop(arg::args) =
            let
              params_inner,vars,types,params_outer = loop(args)
            in
              if arg = pair(head,tail)
                then
                  Lvar(var_tmp)::params_inner,
```

```
                    var_tmp::vars,
                    Obj::types,
                    arg::params_outer
                  else
                    Lit(arg)::params_inner,
                    vars,
                    types,
                    params_outer
          params_inner,vars,types,params_outer = loop(args)
          c = closure(tag_closure/new,tag_lambda/new,∅,vars,types,Lapp(Lit(func)::inners))
       in
         apply_mappable(map_list,closure::params_outer,C)
   else
     error("function arguments do not match")
```

The $\texttt{automap}_{\text{array}}$ function works nearly exactly like its counterpart for lists; only dimension checking is added.

3.7 Top-Level Evaluation

The "top level" evaluation of a program proceeds in two steps: first, the program is evaluated within an initial context, then the scheduler is applied to the resulting state. The scheduler loops and generates a sequence of states as its output, terminating when there are no possible successor states.

3.7.1 $\texttt{eval}_{\text{program}}$

```
eval_program(Lprog(decl,expr)) =
    let
      S_0 = (∅, -1)
      C_0 = (∅, ∅, S_0)
      res_init =
          eval_decl(decl,C_0)
              λ(E_s,E_d,C).
                  eval_expr(expr,C+E_s+E_d+time_0)
      (res,S_1) = norm_result(res_init)
    in
      S_1 :: schedule(S_1)
```

$\texttt{S}_0$ is the initial empty state. It contains no reference definitions; its time is set to an invalid value. This helps to implement a restriction outlined in the informal language definition that is not explicitly modelled in the semantics: the *decl* should not change the state, i.e., $\texttt{S}_1\texttt{=}\texttt{S}_0$, modulo non-run-time definitions.

Before the **expr** can be evaluated, a valid initial $\texttt{time}_0$ is added to the context. From that moment on, suspensions can be created.

3.7.2 schedule

The scheduler is the entity that "drives" the evaluation once the initial suspensions have been created. The scheduler generates a list of states. There is some non-determinism involved; this is modelled in the "probability" of **Wsusp**-selection.

The timing model of the scheduler is a quasi-statical one. It is assumed that all expressions evaluation in zero time; the scheduler manages a "virtual clock" that is incremented stepwise (by $\texttt{advance}_{\mathrm{time}}$) at certain points in the schedule.[29] In the generated code, an increment in time corresponds to a synchronization, i.e. waiting for that time. It cannot be guaranteed that the real execution timing will adhere to the idealized timing presupposed by the scheduler.

The idealization of zero-time evaluation is a necessity; the alternative would need some "real" execution time model. Instead, the ordering of suspensions is chosen in such a way that there is a maximum of "slack" between suspensions of different time: when more than one suspension can be evaluated at a certain point of time, the suspension with the least slack is done first.

The semantic function `schedule` does not represent this strategy explicitly; it rather gives a non-deterministic description of all possible schedule sequences, i.e. all sequences that adhere to the timing specification of the user program. By weighting the specifications with a probability, the scheduling strategy is implied.

```
schedule(S) =
    if ¬ ∃ tag: S[tag] ∈ Ususps ∨ S[tag] ∈ Wsusps ∨ S[tag] ∈ Blocks
      then
        [S]
    [] ∃ tag: S[tag] = Block(tag',cont_blocked) ∧ S[tag'] = EvRef(v)
      then
        case norm_result(cont_blocked(S))
          of (obj,S') => S::schedule(S'[tag/EvRef(obj)])
    [] ∃ tag: S[tag] = Wsusp(expr,time(t_1),time(t_2)) ∧ t_1≤ 0
      then with a probability of 1/(t_2+1)
        case eval_thunk(expr,S)
          of (obj,S') => S::schedule(S'[tag/EvRef(obj)])
    [] ∃ tag: S[tag] = Ususp(expr,cond,time(t_1),time(t_2))
              ∧ eval_thunk(expr,S) = (true,S')
      then
        S::schedule(S[tag/Wsusp(expr,time(t_1),time(t_2))])
    else
      S::advance_time(S)

advance_time(S) =
    decrement all time values in Wsusp
        and increment the system time, generating S';
    perform_io(S')

perform_io(S)
 ∀ tag : S[tag] = Event(parameters) ∧ Oracle(parameters) :
           S' = S[tag/EvRef(ValueOfIO(parameters))]
 schedule(S')
```

The first case handles termination, which happens when there are no suspensions or blocks left. The second case looks for evaluable blocked processes and, if it finds some, chooses one to evaluate. The third case looks for waiting suspensions that may be evaluated in the current time frame. From the set of possible `Wsusp`s, one is chosen with a probability that is inverse to its "slack", i.e. the time range in which evaluation has to take place. If the slack is zero, the suspension will definitely be evaluated. The last case looks for suspensions that have become evaluable in the current time frame; it promotes them to `Wsusp` status. If none of the cases match, time is advanced. As a side-effect of $\texttt{advance}_{\mathrm{time}}$, the time ranges of `Wsusp`s are decremented, and some pending I/O operations are performed. I/O is implementation-dependent; this is

[29] The "step size" is arbitrary; it is an artefact of the specification that needs discrete time. Each actual implementation will declare a smallest possible time unit.

modelled by an oracle function that chooses the I/O events that are to be performed, and by the unspecified function `ValueOfIO` that determines the value of the `EvRef` that replaces an `Event`.

Suspensions become evaluable if their time ranges are shifted into the "now" point, or if blocks are resolved that were caused by I/O events.

The ordering of cases introduces some sequentiality that is not strictly necessary. The `Block` case precedes the general `Wsusp` case, since blocking suspensions are implicitly timed as "immediate", i.e. have to be executed in any case.

By permitting I/O only on time advances, it can be guaranteed that the set of operations accessing a single I/O device can be controlled. The natural semantics of I/O devices is that they can (at most) produce or consume one value per time unit; it is therefore an error if more than one value is written to a device simultaneously; likewise, all read accesses performed at one point of time should return the same value. These conditions are not explicitly tested for, though.

3.8 Discussion of Selected Problems

This section shows the problems associated with selected parts of the semantics.

3.8.1 Macros

The exact semantics of macros was not mentioned in the previous sections. In the current combination of transformation-rules and semantic functions, macros are eliminated during the transformation from the abstract ALDiSP program tree to the LF. As already mentioned, it would be easy to introduce a semantic type that denotes "macro objects" as "call-by-name closures" and a primitive `mk_macro` that transforms a closure into a macro. The evaluation mechanism could be modified so that the functional position of an application is evaluated first. In fact, such an experimental change was made to the reference implementation of the standard semantics. Some annoying problems came up:

- What is the *type* of a macro-object? It is not a procedure.
- Can macros be overloaded?
- Are macro parameters typed?
- Is a macro closed over its environment? Consider the following fragment, in which the `+` operator is redefined:
  ```
  macro f(a,b) = (a+b)*(a-b)
  func a+b = a-b
  func g(x) = f(x+1,x-1)
  ```
  Now, using textual expansion for macros, these definitions are equivalent to
  ```
  func a+b = a-b
  func g(x) = ((x+1)+(x-1))*((x+1)-(x-1))
  ```
  Under a macro-as-run-time-object implementation of macros, which of the "`+`" will be applied to the arguments? The macro-closure will provide one definition for "`+`" (and `*` and `-`); the closure of `g` will provide a different definition. It seems "natural" to use the `g`-definition; but then the other horn of the dilemma presents itself: where does the definition of "`*`" come from? Since a closure environment holds only bindings for those variables occuring free in the closure body, no binding for "`*`" will be found. A static search for macro applications in the body of a closure is also impossible, because the macro may have been given to the function as a parameter! The consequence would be that closures must

save their total creation-time environment. For a variety of reasons, not the least being the problem of garbage collection[30], this approach is unsatisfactory.

To obviate these problems, all macro occurrences are resolved by textual substitution during the transformation phase.

3.8.2 Auto-Mapping

The auto-mapping semantics in the current semantics is much simpler than the one employed in earlier definitions, since it is not as general as possible: It does not include "retrys", i.e. occurrences of more than one mappable type amongst the arguments, when only one combination is correct. An example expression that demonstrates such behaviour is:

```
let
  func t(v:List,x:Integer) = length(v) + x
in
  t(#(1,2,3),#[17,18])
```

Here, $\texttt{automap}_{\text{list}}$ will generate a mapping

```
map_list(λ(w). map_vector(λ(v). t(w,v), #[17,18]), #(1,2,3))
```

and fail. Under a general auto-mapping definition, a mapping would be found:

```
map_vector(λ(v). t(#(1,2,3),v), #[17,18])
```

Under the current implementation, the example expression is erroneous. This is considered a feature: Auto-mapping incurs the pragmatic problem that it might find an interpretation for an application that was never intended by the programmer[31]. If auto-mapping were to go to all lengths to find a mapping, this problem would grow beyond need.

3.8.3 Blocking

A related problem arises from the combination of automatic dereferencing and blocking: The semantics is infested with occurrences of `deref_objs` and related functions that ensue the dereferencing of strict parameters. On the other hand, there are special case treatments for situations in which said dereferencing is to be avoided (type-checking against `Obj`, for example). The semantics would be much simplified if the programmer had to call a function "`deref`" whenever the value of a suspension or promises was needed.

The real problem behind the dereferencing is the blocking behaviour that follows an access to an unevaluated suspension. For this to work, an explicit continuation of the current interpreter state has to be created and *reified*. Reification is the process of lifting semantics-level objects into the object language. It creates a serious follow-up problem: because there is no way to "look into" the reified continuation function, it not possible to compare two such continuation functions, and they cannot be traversed by a garbage collector. The consequences of this problem are are expounded in section 7.4.

Earlier semantics did not use a reification mechanism, but created suitable LF expressions that represented the continuation expression. While that approach solved the comparability and GC problems, it blew up the size and complexity of the semantics, and was hard to debug.

[30] Each closure would then bind a large number of "junk" items which the garbage collector could not dispose of.

[31] This might be called the "TECO problem": when nearly any string of characters is a valid command sequence, the compiler will not catch many errors.

3.8.4 *Implementing Recursion*

The definition of the recursion primitive **fix** as given be the semantics is not very efficient. There exists a "cheap" alterative that employs the existence of state:

```
eval_decl(Lfixd(decls),C) K =
   let
       [v_1,...,v_n] = vars_defined(decls)
       [t_1,...,t_n] = fresh tags
       C' = C[∀ i∈[1...n]:t_i / EvRef(⊥)]
             [∀ i∈[1...n]:v_i / Ref(t_i)]
   in
       eval_pard(decls,C')
           λ(C'',E_s',E_d').
                K(C'''[∀ i∈[1...n]:t_i / EvRef((E_s'∪ E_d')v_i)],
                  E_s',E_d')
```

Each variable defined in the recursive context is given a reference as its definition. The reference is initially defined to be $\perp$; if there is an access to it during the evaluation of the declarations, this is erroneous. Then, all declarations are evaluated in parallel. The resulting values are then stored in the references as the final values. This method has two disadvantages: Each successive access to one of the recursive values is indirected via the reference; this costs time. A bigger problem is the increased size of the state; the state comparison that is an expensive part of the abstract scheduler might be slowed down.

Chapter 4

Transformation of the AST into Lambda Form

This chapter shows how the abstract syntax tree (AST) that is generated by the parser is transformed into an LF program. This transformation provides for an indirect semantics of ALDiSP, since LF programs have a defined meaning, and the transformation rules are total and unique.

The transformation is partitioned into one one major transformation function that generates a "rough translation", followed by a sequence of small local transformation rules that "polish" the LF program.

The three main transformation functions $\mathtt{T}_{\mathrm{program}}$, $\mathtt{T}_{\mathrm{decl}}$, $\mathtt{T}_{\mathrm{expr}}$ and $\mathtt{T}_{\mathrm{type}}$ are of type

$\mathtt{T}_{\mathrm{program}}$: `Aprogram` $\rightarrow$ `Lprog`
$\mathtt{T}_{\mathrm{decl}}$: `Adecl` $\rightarrow$ `Ldecl`
$\mathtt{T}_{\mathrm{expr}}$: `Aexpr` $\rightarrow$ `Lexpr`
$\mathtt{T}_{\mathrm{type}}$: `Atype` $\rightarrow$ `Lexpr`

The transformations are specified to be context-insensitive; they translate AST expressions (generated by the parser, cf. figure 2.3) into equivalent LF expressions without referring to their lexical context or other "compile state". This approach was chosen for simplicity, but incurs some inefficiencies. These inefficiencies are removed by the post-processing phases.

These transformation functions do not take as parameters any kind of environment, continuation expressions, or other "compile state". Since some amount of "global" data must be collected (e.g., the tags that identify abstract data type constructors), a global data base is used to store such information.[1]

The most important post-processing step is macro expansion. It is performed by a LF tree traversal that collects all macro definitions and removes them from the LF program; then all macro occurrences are substituted. To minimize scoping problems, this happens after α-conversion. In the LF program generated from $\mathtt{T}_{\mathrm{program}}$, all ALDiSP macro declarations have been flagged with a `mk_macro` primitive; each definition of the form

```
macro a(x,y) = ...
```

has been transformed into

```
a = mk_macro(lambda(x,y)...)
```

This intermediate representation was chosen to simplify post-processing, which has to find all macro definitions, and expand all macro applications.

[1] This does not violate the compositional structure of the transformation, since these global information are all either sources of unique names, or single-definition associative arrays.

Most post-processing transformations are semantic-preserving LF-to-LFfunctions. Some of them (dead-code removal, α-conversion, detecting unnecessary casts and checks) are local clean-up optimizations that can also be applied later in the compiler; others (lambda hoisting, macro substitution) are more expensive global optimizations that need to be performed only once.

4.1 Transforming Programs: $T_{program}$

The top-level transformation rule is $T_{program}$:

```
Tprogram(Aprogram([ds1,...,dsn],[dr1,...,drm],expr)) =
    Lprog(Lseqd([Tdecl ds1,...,Tdecl dsn]),
          Let(Texpr expr,
              Lfixd([Tdecl dr1,...,Tdecl drm])))
```

The two declaration lists have to be treated differently: the first is to be interpreted sequentially, the second one (representing the `net` declarations) is recursive. The distinction is made explicit by encapsulating the declarations within `Lseqd` and `Lfixd` declarations, respectively. Since recursive definitions are more "expensive" at the implementation level, all `Lfixd` definitions are "sequentialized" in a post-processing step.

4.2 Transforming Declarations: T_{decl}

In ALDiSP, everything of interest is a declaration. The Lambda Form needs only one basic declaration form, plus three declaration combinators (sequential, parallel, and recursive combination). The T_{decl} functions have to decompose the sometimes baroque ALDiSP declaration syntax into these four declaration forms, plus `lambda` and type-checking nodes. At the ALDiSP level, lexical and exception bindings are distinct forms; at the LF level, a boolean flag distinguishes between them. Since only two ALDiSP forms introduce exceptions (`guard` bindings and parameter declaration that are qualified with an explicit `exception` keyword), most transformation rules set the first parameter of `Ldecl` forms to `true`.

4.2.1 `Asimpledecl`

There are basically two cases of simple declarations, overloaded and non-overloaded. To implement overloading, a primitive function `overload` is employed. `overload` takes two arguments that must be closures or overloaded functions and creates a new overloaded function from them.

```
Tdecl(Asimpledecl(overloaded,var,expr)) =
    if overloaded
      then
        Ldecl(true,var,Lapp([Lit(overload),
                             Texpr expr,
                             Lvar(var)]))
    else
        Ldecl(true,var,Texpr expr)
```

4.2.2 `Aparamdecl`

Declarations with parameters are hard to transform, especially in the context of `recs`.

At the AST level, the concept of "parametric declaration" is somewhat confused: each declaration that is preceded by a type keyword (`func`, `proc`, `type`, `exception`, or `macro`) is parsed as an `Aparamdecl`. There is no requirement for the declaration to have a parameter list.

An example of such a "degenerated declaration" is

```
func add = '+'
```

The type keyword forces a type check; the declaration shown above is equivalent to

```
add = [Function] '+'
```

A second problem area is recursion. At the ALDiSP level, a special `rec ... end` construct marks mutually recursive definitions. `rec` is only needed for mutually recursive functions, since all functions are supposed to be recursive "in themselves". This does not have to be handled by the T_{decl}; the parser has already encapsulated simple non-degenerate `Aparamdecl`s into a `recs`.

A post-processing stage is necessary to rename "old" definitions of overloaded variables. Consider the case of a simple overloaded recursive definition. A definition of the form

```
overloaded func f(...) = ...f(...)...
```

has to be transformed into a cluster of definitions

```
Lseqd([ f_old = f,
        Lfixd([
               f = overload(f_old,λ(...) ...f(...)...)
              ])
      ])
```

so that the old name is not shadowed by the recursive definition. This transformation cannot be done at the `Aparamdecl` level, since it depends on the context: the enclosing `Lfixd` can be many nesting levels removed.

The transformation rule for `Aparamdecl` naïvely applies the `overload` primitive if the flag is set, and otherwise cares only about transforming the parameter lists into nested `Lambda` forms. The rule is somewhat complicated by the fact that ALDiSP syntax allows for multiple argument lists; a `Aparamdecl` therefore has as its primary argument a list of parameter lists. The auxiliary function T_{params} creates one nested `Lambda` expression for each parameter list.

```
T_decl(Aparamdecl(overloaded,header,[params_1,...,params_n],var,expr)) =
    let
        (static,type,macro) =
            case header
              of Afunc      => (true ,Type_Function ,false)
              [] Aproc      => (true ,Type_Procedure,false)
              [] Atype      => (true ,Type_Type      ,false)
              [] Aexception => (false,Type_Obj       ,false)
              [] Amacro     => (true ,Type_Obj       ,true)
        expr'  = T_params([params_1,...,params_n],expr)
        expr'' = Lcast(Lit(type),expr',Lvar(cast_general),Lvar(cast_base))
        expr'''= if macro
                   then
                     Lapp([Lit(mk_macro),expr''])
                 else
                     expr''
    in
      Ldecl(static,var,
            if overloaded
```

```
              then
                Lapp([Lit(overload),
                      expr''',
                      Lvar(id)])
              else
                expr'''
```

```
T_params([],expr) = expr
T_params([Aparam(p_1,t_1),...,Aparam(p_n,t_n)]::params,expr) =
    Lambda(tag_new,[(p_1,T_type t_1),...,(p_n,T_type t_n)],
           T_params(params,expr))
```

The list of header cases filters out the type of the declared object, whether it is a macro or not, and whether it is to be bound dynamically or lexically.

The type is used to generate an **Lcheck** that encapsulates the expression. If the expression is a macro, a primitive **macro** is applied to the type-checked expression. The types $\texttt{Type}_{\mathrm{Function}}$, $\texttt{Type}_{\mathrm{Procedure}}$ and $\texttt{Type}_{\mathrm{Type}}$ are not mentioned explicitly in the previous chapter; it is assumed that they are present as primitive functions. The type $\texttt{Type}_{\mathrm{Obj}}$ is the same as **Obj**.

4.2.3 Amultidecl

Multiple-valued declarations are modelled via tuples that are explicitly dismembered; a declaration of the form

```
(v_1,...,v_n) = expr
```

is translated into LF code of the form

```
var_tmp = expr
v_1 = select(1,var_tmp)
  ⋮
v_n = select(n,var_tmp)
```

One new temporary variable ($\texttt{var}_{\mathrm{tmp}}$) is introduced for each **Amultidecl** binding.

```
T_decl(Amultidecl(overloaded,[var_1,...,var_n],expr)) =
    let
      ∀ i∈ 1..n:
          select_i = Lapp([Lit(select),Lit(i),Lvar(var_tmp)])
          decl_i = Ldecl(true,var_i,
                         if overloaded
                           then
                             Lapp([Lit(overload),select_i,Lvar(var_i)])
                         else
                           select_i)
    in
      Lseqd([Ldecl(true,var_tmp,T_expr expr),
             decl_1,...,decl_n])
```

The $\texttt{select}(n, t)$ primitive extracts the n-th element from a tuple t. $\texttt{select}_{\mathrm{i}}$ is the expression that extracts the i-th element from **expr**, and $\texttt{decl}_{\mathrm{i}}$ is the declaration for $\texttt{var}_{\mathrm{i}}$.

4.2.4 Amoduledecl

Modules are implemented as functions that enclose a name space.[2] The encapsulated values are accessed via an `Lselect` expression that uses string representation of the exported names to find the matching object.

The "$\texttt{export}_i$ = `Aexport(...)`" phrase is used to decompose the i-th export clause.

```
Tdecl(Amoduledecl(var_module,[export_1,...,export_k],[decl_1,...,decl_n]) =
  let
    ∀ i∈ 1..k :
        export_i = Aexport(old_i,new_i,type_i)
    expr_select =
        Lselect(Lvar(var_new),
                [(Lit(string(new_1)),Lcheck(T_type type_1,Lvar(old_1)))
                 ⋮
                 (Lit(string(new_k)),Lcheck(T_type type_k,Lvar(old_k)))
                ],
                Lit(⊥))
  in
    Ldecl(true,var_module,
          Lambda(tag_new,[var_new,Lit(String)],
                 expr_select,
                 Let(Lseqd([T_decl decl_1,...,T_decl decl_n]))))
```

The list of exported names is transformed into a list of alternatives of an `Lselect`. An export declaration `Aexport(`*old*, *new*, *typed*`)` exports the object declared originally as *old* under the new name *new* and imposes the *type* on it. If no value is matched, ⊥ is returned.

The function `string` converts a variable name into its string representation.

4.2.5 Aimportdecl

An object is extracted from a module by application of its name string to the module function, i.e.

```
import A as B from X end
```

is translated into code that corresponds to

```
B = X("A")
```

An `Aimportdecl` is transformed into a sequence of declarations. The only complexity is the added type-checking and the ubiquitous handling of overloading.

The "$\texttt{import}_i$ = `Aimport(...)`" phrase is used to decompose the i-th import clause.

```
Tdecl(Aimportdecl(var_module,[import_1,...,import_n])) =
    let
     ∀ i ∈ 1..n:
        import_i = Aimport(overloaded_i,old_i,new_i,type_i)
        access_i = Lcheck(T_type type_i,
                          Lapp([Lvar(var_module),
                                Lit(string(old_i))]))
```

[2] An alternative method would be the use of some textual means to distinguish module scopes. Such a method is considered for the next implementation, since it is expected that it lowers the costs and simplifies some analyses.

```
        access_i'= if overloaded_i
                     then
                       Lapp([Lit(overload),access_i,Lvar(new_i)])
                   else
                     access_i
        decl_i = Ldecl(true,new_i,access_i')
    in
      Lpard(decl_1,...,decl_n)
```

The clauses of an import declaration are grouped into a set of parallel declarations. If the declarations were sequentialized, no warning could be given if a name is multiply defined.

4.2.6 Aabstypedecl

Each datatype declaration is translated into a set of functions to create tagged tuples, to access their contents, and to check their types. A declaration like

```
abstype BinTree = Leaf | ValuedLeaf(value)
                       | Node(left:BinTree,right:BinTree)
```

introduces a set of names and objects:

- constructors `Leaf`, `ValuedLeaf` and `Node`
- selectors `value`, `left` and `right`
- a type `BinTree`, implemented as a predicate

In addition, the range of the `is` predicate has to be extended to recognize the data objects created by the constructors, as in

```
func countLeafs(x:BinTree) =
  if x is Leaf then 1
  [] x is ValuedLeaf then 2
  else countLeafs(left(x))+countLeafs(right(x))
  end
```

This is handled via a global list of constructors that associates constructor functions with tuple tags. To make this comparison "secure", it becomes necessary to *uniquely identify* the constructors. While it is possible to identify constructors by comparing their bodies, this is both inefficient and error-prone, since later phases of the compiler change the definitions of functions. Instead, the constructors are identified with their tags – indeed, this is the very reason for the introduction of these tags in the first place.[3]

4.2.6.1 Parameters

A translation problem comes when constructor parameters typed: The `abstype` declaration can take arbitrary parameters, as in

```
abstype BinTreeOf(X) = Leaf(value:X)
                     | Node(left:BinTree(X),right:BinTree(X))
type IntTree = BinTreeOf(Integer)
type BoolTree= BinTreeOf(Bool)
```

[3] Once lambda- and closure-tags are introduced, they prove to be convenient in a number of places where objects have to be sorted, compared, and cached.

How are constructors and type tests to behave? Imagine a context where the above given declarations are visible, and the following constructors applications are executed:

```
x = Leaf(12)
y = Leaf(true)
z = Leaf(13.3)  \\ error?
```

There are two issues here:

- is the third applications an error? There is no "instantiated type" corresponding to the constructor. On the other hand, the type `BinTreeOf(Float)` can be created somewhere else (i.e., later) in the program.
- Should the "type" of `x` be different from the "type" of `y`? That is, should the `Leaf` constructor be sensitive to the type of its argument and incorporate them into the tuple tag?

  If this is *not* done, a type test like `IntTree(x)` will have to traverse the entire tree to look at the types of all nodes!

  On the other hand, the `Leaf` constructor cannot know all possible types it will applied to later in the program. This is especially true when predicate types occur.

In the current version of the transformation rules, the following strategy is employed:

- If a constructor takes an argument that is restricted to be of type $\texttt{expr}_\mathrm{T}$, and $\texttt{expr}_\mathrm{T}$ contains one of the parameters to the type declaration, no type restriction is built into the constructor.
- For each constructor, a test function is created that is parameterized by an argument list identical to that of the `abstype`. This test function will check the types of all constructor arguments that were dismissed due to the previous rule.

4.2.6.2 An Example Transformation

What follows is an example translation, in which all problem cases that might occur in am `abstype` declaration are exemplified.

```
abstype TestType(P) = ALeaf(p1 : foo(P))
                    | BLeaf(q1 : int,p1)
```

The declarations generated are, in pseudo-code:

```
ALeaf = λ(p1:Obj). _tuple(tag_A,p1)
BLeaf = λ(q1:int,p1:Obj). _tuple(tag_B,q1,p1)
p1 = λ(x:Tuple(tag_A,Obj)) = select(1,x)
q1 = λ(x:Tuple(tag_B,Obj,Obj)) = select(1,x)
p1 = overload(λ(x:Tuple(tag_B,Obj,Obj)) = select(2,x),
              p1)
is_ALeaf(P)(x) = x is ALeaf ∧ foo(P)(p1(x))
is_BLeaf(P)(x) = x is BLeaf
TestType(P)(x) = is_ALeaf(P)(x) ∨ is_BLeaf(P)(x)
```

The `is` function will transform into a simple tag check, e.g.

```
x is ALeaf
```

translates into

```
Tuple(tag_A,Obj)(x)
```

Since the constructor parameter of `is` does not have to be known at translation time, a global list is maintained that associates constructor tags with their tuple types. $\texttt{Tuple(tag,t}_1\texttt{,...,t}_\mathrm{n}\texttt{)}$ is the type of all n-tuples with tag `tag` and component types $\texttt{(t1,...,t}_\mathrm{n}\texttt{)}$.

4.2.6.3 The Rules

```
T_decl(Aabstypedecl([params_1,...,params_p],var_type,[constructor_1,...,constructor_n])=
  let
    vars_params([A_params(v_1,t_1),...,A_params(v_n,t_n)]) = {v_1,...,v_n}
    vars = vars_params(params_1) ∪ ... ∪ vars_params(params_i)
    ∀ i∈ 1...n:
        constructor_i = Aconstructor(id_i,[(p_1,t_1)...,(p_k,t_k)])
        ∀ j∈ 1...k:
             type_j = T_type t_j
             type_intrinsic/j'= if free(type_j) ∩ vars ≠ ∅
                                  then
                                    Lit(Obj)
                                else
                                  type_j'
             type_parametric/j = if free(type_j) ∩ vars ≠ ∅
                                   then
                                     type_j'
                                 else
                                   Lit(Obj)
             sel_j = Lambda(tag_new,[(var_sel,expr_type/tuple)],
                           Lapp([Lit(select),Lit(j),Lvar(var_sel)]))
             decl_sel/j = Ldecl(true,p_j,sel_j)
        expr_type/tuple = Lapp([Lit(tuple_type),Lit(tag_i),type_intrinsic/1',...,type_intrinsic/n'])
        decl_sels/i = Lpard([decl_sel/1,...,decl_sel/k])
        cons_i = Lambda(tag_cons/i,[(p_1,type_intrinsic/1'),...,(p_k,type_intrinsic/k')],
                       Lapp([Lit(_tuple),Lit(tag_i),Lvar(p_1),...,Lvar(pl)]))
        decl_cons/i = Ldecl(true,id_i,cons_i)
        test_i = Lapp([Lapp([Lit(mk_tuple_type),Lit(tag_i),
                            test_parametric/1,...,test_parametric/k]),
                      Lvar(var_test)])

        (side effect : add (tag_cons/i,tag_i) to the list-of-constructors)

    test_type = T_params([params_1,...,params_p],
                         Lambda(tag_new,[(var_test,Lit(Obj))],
                                Lapp([Lit(or), test_1,...,test_n])))

  in
    Lseqd([Lpard([decl_cons/1,...,decl_cons/n,
                  decl_sels/1,...,decl_sels/n]),
           Ldecl(true,var_type,test_type)])
```

The T_{params} transformation is defined in the context of the `Aparamdecl` transformation. The primitive `or` is the logic "or" function, extended to an arbitrary number of arguments.

The "list of params" is the aforementioned global structure that implements the `is` primitive.

4.2.7 `Arecdecl`

Recursive declarations can be turned directly into `Lfixd` forms:

```
T_decl(Arecdecl([decl_1,...,decl_n]))=
    Lfixd([T_decl decl_1,...,T_decl decl_n])
```

In a post-processing step, recursive declarations are simplified further (cf. section 4.5.3).

4.3 *Transforming Type Expressions:* $\mathrm{T}_{\mathrm{type}}$

Transforming a type results in an expression denoting a predicate. There are two kinds of type expressions, function types and "everything else". ALDiSP does not support special syntax to distinguish between type expressions and "value" expressions; the **Aexprtype** is therefore needed to lift ordinary expressions into the syntactic type domain.

4.3.1 Afunctype

This special expression models the special syntactic form ($\mathtt{t}_1$,$\mathtt{t}_2$ -> $\mathtt{t}_3$,$\mathtt{t}_4$) that expresses the types of functions. The transformation generates a call to the primitive **mk_proctype** that creates a *Proc* type object.

$\mathrm{T}_{\mathrm{type}}$(Afunctype([$\mathtt{a}_1$,...,$\mathtt{a}_n$],[$\mathtt{r}_1$,...,$\mathtt{r}_m$])) =
    Lapp([Lit(mk_proctype),$\mathrm{T}_{\mathrm{type}}$ $\mathtt{a}_1$,...,$\mathrm{T}_{\mathrm{type}}$ $\mathtt{a}_n$,$\mathrm{T}_{\mathrm{types}}$($\mathtt{r}_1$,...,$\mathtt{r}_n$)])

Since there are no multiple-valued functions at the LF level, a tuple type has to be created for $m > 1$:

$\mathrm{T}_{\mathrm{types}}$([x])= $\mathrm{T}_{\mathrm{type}}$ x
$\mathrm{T}_{\mathrm{types}}$([$\mathtt{x}_1$,...,$\mathtt{x}_n$]) =
    Lapp([Lit(tuple_type),$\mathtt{tag}_{\mathrm{multi}}$,$\mathrm{T}_{\mathrm{type}}$ $\mathtt{x}_1$,...,$\mathrm{T}_{\mathrm{type}}$ $\mathtt{x}_n$])

A special tag $\mathtt{tag}_{\mathrm{multi}}$ is reserved for multi-return values.

4.3.2 Aexprtype

An "expression type" is an ordinary expression:

$\mathrm{T}_{\mathrm{type}}$(Aexprtype(expr)) = $\mathrm{T}_{\mathrm{expr}}$(expr)

No check is made the **expr** really denotes a type value or a predicate.

4.4 *Transforming Expressions:* $\mathrm{T}_{\mathrm{expr}}$

The next few transformations model one-to-one correspondences between ALDiSP- and LF-expressions.

4.4.1 Avar

$\mathrm{T}_{\mathrm{expr}}$(Avar(var)) = Lvar(var)

No distinction is made between dynamic and lexical variable lookup at either the ALDiSP or the LF level.

4.4.2 Aapp, Acond, Acheck

$\mathrm{T}_{\mathrm{expr}}$(Aapp([$\mathtt{expr}_1$,...,$\mathtt{expr}_n$])) =
    Lapp([$\mathrm{T}_{\mathrm{expr}}$ $\mathtt{expr}_1$,...,$\mathrm{T}_{\mathrm{expr}}$ $\mathtt{expr}_n$])

$\mathrm{T}_{\mathrm{expr}}$(Acond($\mathtt{expr}_1$,$\mathtt{expr}_2$,$\mathtt{expr}_3$)) =
    Lcond($\mathrm{T}_{\mathrm{expr}}$ $\mathtt{expr}_1$,$\mathrm{T}_{\mathrm{expr}}$ $\mathtt{expr}_2$,$\mathrm{T}_{\mathrm{expr}}$ $\mathtt{expr}_3$)

$\mathrm{T}_{\mathrm{expr}}$(Acheck(type,expr)) =
    Lcheck($\mathrm{T}_{\mathrm{type}}$ type,$\mathrm{T}_{\mathrm{expr}}$ expr)

4.4.3 Acast

```
T_expr(Acast(type,expr)) =
    Lcast(T_type type,T_expr expr,Lvar(cast_general),Lvar(cast_base))
```

The cast expression has two "invisible" parameters `cast_general` and `cast_base`. These variables define overloaded functions that implement the cast functionality. They are provided by a system library and can be modified by the user. The `cast` only serves as a generalized "dispatch" so that the programmer does not have to remember the names of dozens of conversion functions.

4.4.4 Adelay

```
T_expr(Adelay(expr)) =
    Lapp([Lit(_delay),Lambda(tag_new,[],T_expr expr)])
```

The primitive function `_delay` transforms a thunk[4] into a `Ususp`. A new, parameterless lambda expression is generated to encapsule the delayed expression.

4.4.5 Asuspend

```
T_expr(Asuspend(expr,cond,t_1,t_2)) =
    Lapp([Lit(_suspend),Lambda(tag_new,[],T_expr expr),
                        Lambda(tag_new,[],T_expr expr),
                        T_expr t1,T_expr t_2])
```

This works analogously to the `delay` transformation. The two tags are distinct. Both the suspended expression and the conditional expression have to be converted into thunks. There is no syntactic check for the types of `t1` and `t2` (they should be of type "`Duration`"); such tests are deferred to run time.

4.4.6 Atuple

```
T_expr(Atuple([expr_1,...,expr_n])) =
    Lapp([Lit(tuple),Lit(tag_multi),T_expr expr_1,...,T_expr expr_n])
```

Here, the $\texttt{tag}_{multi}$ mentioned in section 4.3.1 resurfaces.

4.4.7 Aseq

```
T_expr(Aseq(stmts)) = T_stmts(stmts)
```

ALDiSP sequences can contain both declarations and statements. Statements (i.e., expressions) are *synchronized*; if a statement evaluates to a suspension, further evaluation blocks until this suspension is available.

Sequences are translate into nested declarations and LF sequences:

```
T_stmts([]) = error("empty sequence")
T_stmts(Adecl(decl) :: stmts) =
    Let(T_decl decl,
```

[4] Traditionally, a function that takes no arguments and is used to delay the evaluation of an expression is called a "thunk". The origin of the term is obscure; it was possibly coined when thunks were first used to implement call-by-name in ALGOL-60 (cf. [48], p. 201).

```
        Tstmts stmts)
Tstmts([Aexpr(expr)]) = Texpr expr
Tstmts(Aexpr(expr)::stmts) = Lseq([Texpr expr,
                                   Tstmts stmts])
```

An alternative translation might do away with LF-level sequences and sequentialize the statements using explicit suspensions. This was not done in `ac` because there are situations in which it is convenient to generate LF sequences. In effect, this transformation is now done by the semantics.

4.4.8 Astring, Aint, Afloat, Aconst

```
Texpr(Astring(x)) = Lit(x)
Texpr(Aint(x))    = Lit(x)
Texpr(Afloat(x))  = Lit(x)
Texpr(Aconst(x))  = Lit(lookupconst x)
```

These transformation rules do not convey the exact transformation. The string/int/float values are injected into the "`Obj`" domain. `Aconst` values (which are symbols just like variables) are looked up in a table of literals. This table contains such values as "`Obj`" and "`true`", i.e. the names of basic objects that are not best described as primitives.

If a constant is not found in the table, it is considered to be the name of a primitive. The set of primitive operations is not fixed; primitives are only looked up by the semantic function that implements their application behaviour.

4.4.9 Aguard

A `Aguard` expression has two purposes: it introduces a set of dynamic bindings, and it catches exceptions. The bindings are modelled using normal declaration transformations. The `Lcatch` form models exception handling.

To construct the `Lcatch` expression, the list of exception tags that can be caught is needed. This list is extracted from the `Aguard` expression by searching the exception declarations for occurrences of the application sequence generated by the `return` statement (`_mk_exc(tag,...)`):

```
Texpr(Aguard([decl1,...,decln],expr)) =
  let
    ∀ i∈1..n:
        di = Tdecl decli
    tags = collect([d1,...dn])
  in
    Let(Lpard([d1,...,dn]),
        Lcatch(tags,Texpr expr))

collect([]) = []
collect(decl::decls) =
   if Lapp([Lit(_mk_exc),Lit(tag),...]) ∈ decl
     then
       tag :: collect(decls)
   else
     collect(decls)
```

4.4.10 `Areturn`

The **`Areturn`** aborts the current evaluation and raises an exception. The exception can be parameterized with arbitrary data values. In the semantics, exceptions are a special result type; the LF code that implements the **`Areturn`** therefore only constructs such a value. No control flow manipulation is needed.

$\mathtt{T}_{\mathrm{expr}}$`(Areturn(expr`$_1$`,...,expr`$_n$`)) =`
    `Lapp([Lit(_mk_exc),Lit(tag`$_{\mathrm{new}}$`),`$\mathtt{T}_{\mathrm{expr}}$` expr`$_1$`,...,`$\mathtt{T}_{\mathrm{expr}}$` expr`$_n$`])`

The primitive **`_mk_exc`** constructs an exception value.

4.4.11 `Alocal`

A local declaration in ALDiSP has exactly the same semantics as a **`Let`** expression in LF. The declarations are implicitly ordered sequentially.

$\mathtt{T}_{\mathrm{expr}}$`(Alocal(expr,[decl`$_1$`,...,decl`$_n$`]) =`
    `Let(`$\mathtt{T}_{\mathrm{expr}}$` expr,`
        `Lseqd([`$\mathtt{T}_{\mathrm{decl}}$` decl`$_1$`,...,`$\mathtt{T}_{\mathrm{decl}}$` decl`$_n$`]))`

4.5 *Post-processing Passes*

After $\mathtt{T}_{\mathrm{program}}$ has been applied to the abstract syntax tree, a number of post-processing passes make small changes to the LF program. Some of these passes are needed to fix problems that were not treated correctly by the transformation rules; others are classical compiler transformations that remove nodes from the program tree or re-arranges them to minimize execution characteristics such as time and space consumption. Not all of these optimizations are implemented in the current version of the compiler.

There is no real "need" for optimizations at this point of the compilation trajectory, as the partial evaluator can work on the un-optimized programs and should generate functionally equivalent code, but there are some practical reasons for including them:

- Since abstract interpretation is much slower than standard interpretation, it is advantageous if the program tree is pruned by removing all excess operations as early as possible.
- Debugging the later stages of the compiler is much simpler if the programs are small.

4.5.1 *Implementation of post-processing steps*

A general tree-walker has been written that collects the lexical environments for all variables while walking the tree; the post-processing transformations are instantiations of this walker. A transformation rule is applied at each matching node in bottom-up order; the current static environment is given as a parameter. In the environment, each variable is either defined in terms of some expression, or marked as bound in a **`Lambda`** parameter list.

4.5.2 *Expanding Macros*

After the main transformation rules have been applied to the abstract syntax tree, all macro declarations are collected and removed from the programs, and the macros are replaced by textual substitution everywhere. No attention is paid to name clashes, i.e., the fragment

```
g = g1
macro f(a,b) = let tmp = g(a,delay b) in h(g,a,b) end
g = g2
tmp = 42
f(x,tmp)
```

will be transformed into

```
g = g1
g = g2
tmp = 42
let tmp = g(x,delay tmp) in h(g,x,tmp) end
```

with all problems involved. ALDiSP macros are not meant to provide hygienic [74, 28] macro expansion.

The expansion function can be described by the local transformation function

```
transform_macro(node,env) =
  case node of
    Lapp([Lvar(x),arg_1,...,arg_n]) =>
      if env[x] = Lapp([Lit(mk_macro),Lambda(tag,[(p_1,t_1),...,(p_m,t_m)],body)])
        then
          retag_lambdas(body[p_1/arg1,...,p_n/argn])
        else
          node
    else
      node
```

One nasty problem is solved by the `retag`$_{\text{lambdas}}$ function: if the `body` contains `Lambda` expressions and the macro is instantiated more than once, the final program will contain multiple `Lambda` expressions with the same tag. `retag`$_{\text{lambdas}}$ walks a tree and gives a new tag to each `Lambda` expression it encounters.[5]

4.5.3 Declaration Simplification

The Lambda Form has three forms of compound declarations: parallel, sequential and recursive. From these building blocks, deeply nested declaration structures can be built. The transformation rules generate many such "groups" that contain only one declaration; the macro-expansion even generates empty groups (to remove a macro declaration by a local transformation, it is replaced by an empty declaration group). One pass therefore compacts declarations; the transformation rule is as follows:

```
simplify_declaration decl =
  case decl of
    Ldecl(s,v,e) => x
  | Lpard(decls) => simplify_pard(traverse_pard [] decls)
  | Lseqd(decls) => simplify_seqd(traverse_seqd [] decls)
  | Lfixd(decls) => simplify_fixd(traverse_fixd [] decls)
```

The three `traverse` functions merge nested declarations of the same type:

```
traverse_pard done todo =
  case todo of
    [] => done
  | (decl::todo) =>
```

[5] This approach could cause real problems if there are references to specific tags, but luckily these occur only in `abstype` declarations. Putting such a declaration into a macro does not make much sense, since this would introduce constructors with multiple definitions. These would not have a well-defined semantics, anyway.

```
      case decl
        of Lpard(decls) => traverse_pard (done @ decls) todo
         | Lfixd([]) => traverse_pard done todo
         | Lseqd([]) => traverse_pard done todo
      else traverse_pard (decl :: done) todo

traverse_seqd done todo =
  case todo of
    [] => done
  | (decl::todo) =>
      case decl
        of Lseqd(decls) => traverse_seqd (done @ decls) todo
         | Lfixd([]) => traverse_seqd done todo
         | Lpard([]) => traverse_seqd done todo
      else traverse_seqd (done@[decl]) todo

traverse_fixd done todo =
  case todo of
    [] => done
  | (decl::todo) =>
      case decl
        of Lfixd(decls) => traverse_fixd (done @ decls) todo
         | Lpard([]) => traverse_fixd done todo
         | Lseqd([]) => traverse_fixd done todo
      else traverse_fixd (done@[decl]) todo
```

$\texttt{simplify}_{\text{fixd}}$ views the `decls` as a set of declarations; it tries to separate it into two independent subsets. If they are found, a parallel declaration consisting of two recursive declarations is created.

```
simplify_fixd(decls) =
   if ∃ decls_1,decls_2: decls_1 ∪ decls_2 = decls ∧ |decls_1| > 0 ∧ |decls_2|>0
       ∧ def(decls_1) ∩ use(decls_2) = ∅
       ∧ use(decls_1) ∩ def(decls_2) = ∅
   then
        Lpard([simplify_fixd decls_1, simplify_fixd decls_2])
   else
        if |decls|=1 then decl_1
                     else Lfixd(decls)
```

$\texttt{simplify}_{\text{seqd}}$ looks for declarations that can be "lifted" from the sequential contexts because they do not refer to previous definitions or provide definitions needed by the following definitions.

```
simplify_seqd([decl_1,...,decl_n]) =
   if ∃ i,k : 1<i<k<n
       ∧ def([decl_1,...,decl_{i-1}]) ∩ use([decl_i,...,decl_k]) = ∅
       ∧ def([decl_i,...,decl_k]) ∩ use([decl_{k+1},...,decl_n]) = ∅
   then
        Lpard([simplify_seqd([decl_1,...,decl_{i-1},decl_{k+1},...,decl_n]),
               simplify_seqd([decl_i,...,decl_k])])
   else
        if n=1 then decl_1
               else Lseqd([decl_1,...,decl_n])
```

$\texttt{simplify}_{\text{recd}}$ replaces one-element `Lrecd` nodes by their contents.

```
fun simplify_pard(decls) =
  case decls
```

```
    of [x] => x
  else Lpard(decls)
```

4.5.4 Removing unnecessary Checks and Casts

During the transformation phase, some `Lcheck` and `Lcast` operations were introduced that have `Obj` as their type. They can be removed safely:

```
remove_cast expr =
  case expr
    of Lcast(Lit(Obj),x,c_general,c_base) => x
     | Lcheck(Lit(Obj),x) => x
  else expr
```

4.5.5 α-Conversion

α-Conversion is the process of giving each statically bound variable a unique name. Once this property has been achieved, it is much easier to rearrange parts of the program, since name clashes are now impossible.

Implementing the α conversion is trivially achieved by an LF syntax tree traversal that keeps track of the lexical environment and introduces appropriate renamings. As a side effect, lexically unbound variables are found, and appropropriate warnings issued when no matching exception definition exists.

4.5.6 Problems with Rearranging Expressions

It would be nice if preprocessing steps would be allowed to rearrange function applications, e.g. to hoist them out of loops, to merge common subexpressions, and to remove dead code. But such transformations are not secure, since in ALDiSP function application may have side effects – it would certainly be deleterious to a program's semantics if an application of `write` would be optimized away or hoisted from a loop, because the result was ignored! Likewise, rearranging the order of side-effects has to be avoided.

Therefore, only those changes to the program are allowed that are guaranteed not to change any side-effects. Amongst the allowed transformations are:

- moving `Lit` expressions,
- moving `Lambda` expressions: they only capture parts of the lexical environment, and can otherwise be treated as literals, i.e., no "action" is involved in their evaluation,
- application of known side-effect-free primitives to arguments that are guaranteed not to be suspended. Every other application may invoke the creation of a direct side effect or a `Block`, and
- moving declarations that contain only side-effect-secure expressions.

4.5.7 Lambda Hoisting

This optimization tries to move each `Lambda` expressions "up" to the point where it is nearest to the definition of its free variables.

One place where this optimization helps is module declarations – since the declarations do not depend upon the selector variable, they can be hoisted to the global level. More opportunities for lambda hoisting will be mentioned in chapter 8.

The implementation of lambda hoisting is much simplified by the preceding α conversion:

```
hoist expr =
  case expr
    of Lambda(tag,params,expr) =>
        let
          lambdas = collect-lambda-expressions(expr)
          defs = collect-Ldecls(expr)
          vars_root = params ∪ params(lambdas)
          needs_vars(vars) = {d | d ∈ defs, free(d) ∩ vars ≠ ∅}
          defs_need = hull_transitive(needs_vars,vars_root)
          defs_hoist= defs - defs_need
        in
          Let(Lambda(tag,params,remove_defs(expr,defs_hoist),
              Lfixd(defs_hoist))
  else expr
```

This transformation is local to each `Lambda` expression. First, all occurrences of `Ldecl` nodes are extracted from the body of the `Lambda`. Then, the set of all declarations that refer to the `Lambda`s parameters is enumerated. The $\texttt{hull}_{\text{transitive}}$ function extends the `needs` function so that both direct and indirect variable dependencies will be found. The root of the dependency search consists of the variables defined in the current `Lambda` node and those defined by all `Lambda` nodes found within the expression.

Finally, all declarations that are found to be independent from the parameters can be hoisted. The hoisting itself is implemented by introducing a new `Let` node that contains a recursive sub-declaration that surrounds the hoisted declarations. The recursivity is introduced since it is not known whether the declarations come from a recursive context or not; due to the α conversion, the `Lfixd` will not "clobber names".

Further simplification transformations will remove unnecessary `Lfixd` decls and empty `Let` nodes (which occur when no hoistable declarations are found).

4.5.8 Transforming Free Variables into Parameters

If a function a has a free variable x, and each call site of a is statically known, x can be passed as an additional parameter to a. This is possible since all call sites of a are only known when it does not "escape" the local scope, which implies that x is visible wherever a is visible. For example, in

```
func b(x,y) =
 let
  func loop(z) = ... x ... loop(z') ...
 in
  loop(y)
 end
```

the variable `loop` cannot escape, since it is not returned as a result. Therefore, the fragment can be rewritten as

```
func b(x,y) =
 let
  func loop'(x,z) = ... x ...loop'(x,z')...
 in
  loop'(x,y)
 end
```

This transformation (which might or might not be an optimization) is not implemented in the current compiler.

4.5.9 Strictness Analysis

Finally, there is a very ALDiSP-specific use of *strictness analysis* (SA): SA tries to determine, for each parameter of a function, if the parameter is accessed in all execution paths of the function. If this is the case, the function is said to be *strict* in this parameter.

SA is mostly employed in lazy languages to find applications that can be called by value without risking nontermination. In ALDiSP, the goal of SA is to avoid blocking: If a function is strict in all its arguments, any suspended argument will block the function eventually. Consider

```
func f(a,b,c) = a+2*(b+3*c)
...
f(1,2,suspend ... end)
```

According to the semantics, a chain of blocks will be created:

```
block1 = 3*c
block2 = b+block1
block3 = 2*block2
block4 = a+block3
```

Strictness analysis would infer that `f` is strict in all its arguments, and would represent this by introducing types:

```
func f(a:Strict(Obj),b:Strict(Obj),c:Strict(Obj)) = a+2*(b+3*c)
```

Any application of `f` with a reference argument would then directly force the block:

```
block1 = f(1,2,suspend ... end)
```

In typical DSP applications, nearly all functions are strict.

By avoiding blocks, the state space created by the abstract scheduler is minimized in advance, which reduces both the run time and the memory consumption of the compiler.

Due to correctness problems, this optimization is not implemented in the current compiler: even if a function is strict in an argument, it might have side effects before that argument is accessed.

Chapter 5

Partial Evaluation – Definitions and Known Results

Before looking into the implementation of the combined abstract interpreter and partial evaluator (AI/PE) that is incorporated in the `ac` compiler, the theoretical underpinnings of PE shall be sketched out. A good summary of the state of the art (1988) can be found in the proceedings of the first workshop on partial evaluation and mixed computation [17]. Recent literature can be found in the proceedings of the *Partial Evaluation and Semantics-Based Program Manipulation* workshops [113].

The first comprehensive "textbook" on PE is [67]. It is mostly concerned with self-applicable off-line partial evaluators. These consist of extensive binding time analyses that annotate the program text with "dynamic/static" tags. These tags guide the following text rewriting phase that inlines and specializes function definitions.

A number of definitions presented in the following pages are quoted from the terminology chapter of [17].

5.1 Definitions

In the following, the semantics of a language L is considered a function

$L : program_L \rightarrow input_{program} \rightarrow output_{program}$

where $program_L$ is the set of L-programs. A language is therefore identified with its interpreter. The semantics of any program is given by its functional behaviour; the semantics of p ($p \in program_L$) can be derived by the application $L\ p$.

5.2 Partial Evaluation, Residual Program

To quote from [17]:

> Partial Evaluation: A process that given a program and part of its input will produce a residual program which when executed with the remaining input of the original program will give the same result as the original program would if it was executed with the complete input. Partial evaluation is also called projection. Partial evaluation can be expressed by the equation
>
> $L\ (L\ mix\ < program, v_1, \ldots, v_n >) < v_{n+1}, \ldots, v_m > = L\ program\ < v_1, \ldots v_n, v_{n+1}, \ldots, v_m >$
>
> where *mix* is a partial evaluator for the language L. A different way of stating the same is that

$resid \;=\; L\ mix\ < program,\ v_1,\ \ldots,\ v_n >$
implies
$L\ resid\ < v_{n+1} \ldots v_m > = \ L\ program\ < v_1, \ldots v_n, v_{n+1}, \ldots, v_m > .$

In analogy to the *residual program*, there are *residual functions* created by specializing single functions with respect to some parameters.

Defined in this way, PE is a program-to-program transformation that does not change the language of the transformed program, but only its eventual run-time costs (and the number of parameters it takes at run-time). Neither time nor space optimization is promised.

The most trivial partial evaluator is the identity function:

$$mix_{id}\ < program, < v_1, \ldots, v_n >> = \ \lambda(v_{n+1}, \ldots, v_m).program\ < v_1, \ldots v_n, v_{n+1}, \ldots, v_m > .$$

5.3 Self-Application and the Futamura Projections

To quote from [17]:

> Self-Applicable Partial Evaluator: A partial evaluator that is written in the same language that it processes, giving the possibility of applying it to itself. Also called an autoprojector.

There are three interesting cases of self-application in PE. First described in [52], they are commonly known as *Futamura projections*. Given a partial evaluator mix_L (written in L and transforming L programs), an interpreter int_L written in L and interpreting some arbitrary language, and a $program_{int}$ written in the language that int_L interprets. Then the first projection

$$L\ mix_L\ int_L\ program_{int} \;=\; program_L$$

is the *specialization of an interpreter* with regard to the program. This equation does not mention the *run-time* inputs of the interpreter (or the interpreted program):

$$L\ (L\ mix_L\ int_L\ program_{int})\ inputs \;=\; L\ program_L\ inputs$$

That is, an interpreter takes two inputs: a program that has to be interpreted and a set of input data for that program.[1]

The result of the PE is a program that has the same semantics as the original interpreted program, but is written in L instead of the *int*-language: A translation has taken place. Again, it shall be mentioned that this translation need not in any way improve the efficiency of the translated program; this equation holds for the identity-PE, too. It just has the opportunity to be efficient; i.e. it is *possible*, but not *necessary* that the run-time cost of ($L\ program_L\ inputs$) are, for some or all inputs, lower than those of ($L\ int_L\ program_{int}\ inputs$). Even if some costs are lower (most likely time), others may well be higher (usually space).

The next projection is

$$L\ mix_L\ mix_L\ int_L \;=\; compiler_L$$

Now, the *mix* specializes *itself* with regard to the interpreter. The result is a compiler, i.e.

$$L\ compiler_L\ program_{int} \;=\; program_L$$

where the $program_L$ is functionally identical to the one generated by the $mix_L\ int_L$ combination. Just as the first projection has the potential of optimizing the cost of running $program_{int}$, this second projection can

[1] Formally speaking, applying the interpreter int_L to a program $program_{int}$ results in a function that takes *inputs*.

optimize the cost of running the compiler. The partial evaluator specializes itself to the task of compiling *int*-programs.

The third and last projection is

$L\ mix_L\ mix_L\ mix_L\ =\ compiler_L^2$

The result is a compiler-compiler. That is,

$L\ compiler_L^2\ int_L\ =\ compiler_L$

Here mix_L is specialized to the task of compiler-generation. There are no more projections, because a "fixed point" is reached: Further self-applications results in the same $compiler_L^2$.[2]

5.4 Internal Specialization

`ac` is an application of the first Futamura projection. There is no separate PE and interpreter; instead, interpreter and PE are mixed into one program. The technical details (and the motivation behind this design decision) are explicated in the following chapters.

The PE employed in `ac` does not implement full partial evaluation, but merely *internal specialization*, as defined in [17]:

> Internal Specialization: Specialization of components of a system to exploit their internal contexts of use while preserving overall system functionality.

5.5 On-line/Off-line Partial Evaluation

Partial evaluators can be differentiated into *on-line* and *off-line* PEs. An off-line PE transforms a given program without actually running it, i.e. it transforms a "passive" program image. On-line PE works by running the program (or simulating its execution) and emitting residual code while doing so. On-line PE is thus a special kind of abstract interpretation where abstract values are generated that contain both value descriptors and program code. Off-line PEs usually consist of two distinct phases: a *binding-time analysis* (BTA), and a specialization phase.

5.6 Static and Dynamic Binding Times

To quote from [17]:

> Binding Time: The time at which a variable/expression is bound to a definite value. In partial evaluation variables bound (known) at partial evaluation time are called static as opposed to dynamic. (Alternatively the terms known and unknown are used)
>
> Binding Time Analysis: An algorithmic analysis that finds approximations of the binding times of variables/expressions in a program.

A simple BTA can be implemented using an abstract interpretation on a two-element data domain $\{\bot, \top\}$. $\top$ models the values that are statically known, $\bot$ those that are dynamic. BTA becomes interesting when *partially static values* have to be considered:

[2] Repeated self-application is indeed a measure for the quality of a PE system. An 'optimal' PE should generate identical copies of itself (modulo renaming) upon self-application.

5.7 Partially Static Values

To quote from [17]:

> Partially Static Value: A structured value that contains both static and dynamic parts.

In ALDiSP, these structured values are mostly lists and higher-order functions. Functions are "data structures" in that they consist of (constant) code and an environment. These environments will contain bindings to static and dynamic values, and to other functions. Annoying problems occur if partially static values can be of potentially infinite size (streams, recursive procedures). Important special cases of partially static data structures are streams. In many stream applications, constant prefixes occur. If these are treated specially, unwanted loop unwinding for the first k might occur. The opposite effect can also create problems: certain optimizations might be impossible when a partially static value is approximated as totally dynamic.

5.8 Specialization

The specialization phase of an off-line PE surveys all function definitions and their call sites and either

- *expands* (unfolds/inlines) the function applications, i.e. substitutes a call site with the function's definition,
- *specializes* a function, i.e. removes a parameter that is bound to a static value in an application, and generates a residual function in which that parameter is replaced by that value, or
- leaves the function call interface as it is.

Both excessive unfolding and specialization can result in *code explosion*, in the worst case nontermination of the PE.

5.9 Generalization

A *generalization* step is often performed to avoid over-specialization: the abstract values encountered in different call sites of the function will be merged so that a more general call interface is found that still leads to the same specialized function body. The main goal here is the minimization of code space.

5.10 Folding/Unfolding

To quote from [17]:

> Unfolding: The substitution of a name by the structure it denotes. Most often used to describe the substitution of a function call with an instance of the definition of the function.
>
> Folding: Upon recognizing that a configuration is similar to a previous configuration, a recursive definition can be made by defining a function or predicate denoting the previous configuration and making the new configuration into a call or reference to that definition. If the two configurations are not exactly identical a generalized definition encompassing both configurations can be made.

While both definitions are formulated in a general way, the "structure" or "configuration" nearly always refers to functions and expressions containing function calls. They could as well refer to data structures, but those are usually directly specified by the user and less malleable.[3]

A program transformation system written by Burstall and Darlington [22, 23] was the first to use the fold/unfold operations as primitives to perform extensive program manipulation. This system established the fact that these two transformations, together with δ-reduction (i.e. application of primitive functions), can be combined to accomplish all possible program transformations.

Unfolding is a generalization of function *inlining*. Most compilers that perform inlining restrict it to very simple functions, e.g. functions that comprise one basic block.

Folding is a generalization of common subexpression elimination (CSE). Here, the "configurations" are expressions, and the "definition" is the introduction of a new variable holding the value of the multiply used definition. While CSE can be done in near-linear time by employing hash-consing strategies [124], generalized folding has a much higher complexity.

5.11 Fixed Points

One of the central problems of AI is the computation of *fixed points* (or *fixpoints*) of recursive functions. In the context of PE this means the modelling of possibly nonterminating executions traces, i.e. dynamically bound loops and recursive function calls. Since an abstract interpreter should terminate for all programs (even programs that are nonterminating for all inputs), a naïve simulation has to be aborted by a time-out mechanism, returning an approximation to the real semantics.

The fixed point theorem suggests a direct way to compute the FP, namely o return $\perp$ the first time a function is called recursively, and to iterate with the result thus obtained until it does not change anymore.

Finding a fixpoint is a very time-consuming process; the worst-case complexity for first-order functions is exponential in the "depth" of the abstract domain [91, 66].

5.12 Applying the Techniques of Abstract Interpretation to ALDiSP

The next chapters will present each of these steps: abstract interpretation, abstract scheduling, code reconstruction, and machine code generation.

At the core of the ALDiSP compiler lies an abstract interpreter for the LF language. This abstract interpreter works on a domain that is detailed enough for constant folding, yet abstract enough to terminate on all input programs. The abstract interpreter is derived from the standard semantics.

The AI consists of two, largely independent, modules. To separate the "pure functional" subset of the language from the "realtime" parts, an *abstract scheduler* handles the set of possible states a program can encounter. To the scheduler, the abstract interpreter is essentially a black-box subroutine that transforms one state into another by evaluating one suspension. The scheduler chooses which suspension when to evaluate, and how to merge states that are similar.

To generate code, the abstract values (which are generated within the abstract interpreter) are given *code attributes*. The idea is that every output value generated by the AI process has a "computational history" attached to it, from which the code necessary to compute it can be generated. These code attributes are not visible to the AI or the AS, i.e. they have no semantic relevance.

[3] In many languages, e.g. C, data structure definitions may not be re-arranged or otherwise changed by the compiler at all.

Based on the code attribute is the *code generation*, which *reconstructs* the program from the code attribute of the abstract results. The reconstruction phase takes the graph of states which was generated by the abstract scheduler, and generates, for each transformation between two states, a piece of code that implements that transformation at run0time. This code (the *Code Form, or CF* is wholly disjoint from the LF code on which the AI was run. It mostly resembles single-assignment dataflow languagess like SILAGE [110, 111] or SISAL [84].

Finally, this CF program is converted into a assembly-level output format (the *Machine Language* or M), which can be translated into any imperative code needed. A sample C back-end has been written.

Chapter 6

Abstract Interpretation of LF Expressions

This chapter introduces the basic mechanisms of abstract interpretation (AI). Using an example, some problems are discussed in detail. The example uses standard functional language notation that corresponds to a functional subset of the Lambda Form. The example is interpreted in an abstract data domain that utilizes interval arithmetic.

In the following, all problems related to the treatment of "state" are ignored. Chapter 7 will show how the abstract scheduler (AS) manages the set of possible states a program can encounter. The abstract scheduler provides the "superstructure" of the compiler; the AI is a "subroutine" of the AS. The AI can ignore the state, since the abstract interpreter is always invoked in a context where the state is fixed. The loop detection (cf. section 6.3.2) ignores functions that change the state; loops that go "across states" are handled by the state loop detection scheme that is part of the abstract scheduler (cf. section 7.5).

6.1 Goals of Abstract Interpretation

The goal of AI is, generally speaking, to acquire information about functions by executing them on abstract data.[1] Information can be gathered both during the execution or afterwards, by analyzing the results. Furthermore, both "forward" and "backward" AI is possible: forward analysis mimics the normal evaluation function; backward evaluation starts with a result and recreates the possible execution traces that might have generated it. Forward AI is the more "natural" approach, since it closely corresponds to normal interpretation. Some analyses are most naturally implemented in a "backward" framework [61].[2] The AI employed in `ac` is of the "forward" kind, since it has been developed by evolutionary means, i.e. by extending the standard semantics. Necessary "backward" optimizations are performed in separate optimization steps (cf. section 8.5).

Different kinds of information can be obtained by AI:

- How often is a function called?
- What are its arguments?

[1] In [67], the term "program point" is used for those programming constructs that are analyzed by an AI. Program points must have a defined argument/result interface. For example, the entry point of a loop construct in an imperative language fits into this category, and an AI could determine how often a loop is entered, or which invariants hold for all loop entries. In ALDiSP, as in most functional languages, user-level functions provide the only program points of interest, since loops are expressed by tail-recursive function application.

[2] Typical "backward applications" are dead-code elimination, the related strictness analysis [60, 118], and compile-time garbage collection [63].

- What does the call stack look like at each invocation?
- How are the arguments used? Is the function strict in all its parameters? Are some arguments never used, or only rarely?
- How much time/space does the function's execution consume?
- Is the function recursive? If yes, is there a bound to the recursion depth?

In general, AI does *not* guarantee to deliver *exact* results: to do so would require excessive (or even infinite) run time, since all possible combinations of arguments to a given function would have to be simulated. Especially in the presence of recursion, AI faces the problem of non-termination. If a possibly non-terminating execution has to be simulated, the abstract interpreter will choose some appropriate *approximation* (or *generalization*) to ensue termination. Such generalizations can be guaranteed to be correct, but are usually imprecise: information will be lost.

In the context of `ac`, the abstract interpreter has the additional task of "computing code": parts of the partial evaluator are implemented within the AI framework, and each abstract value has a `code` component that denotes the CF code needed to compute it at run-time (chapter 8 gives the details).

6.2 Abstract Values

6.2.1 General Remarks

Probably the most important issue in designing an abstract interpretation scheme is the choice of an *abstract value domain*. There are two conflicting goals in choosing this domain:

- The size and run time complexity of the AI will be adversely affected by a highly refined abstract value domain. It is also more work to correctly implement it.
- The precision of analysis will suffer if the abstraction is too coarse.

Each abstract value represents a set of concrete values. One method to characterize an abstract domain is by defining two functions, the *abstraction function* and a *concretization function*. The abstraction function maps the standard data domain to its abstract counterpart, the concretization function does the opposite. The concretization function is usually a relation; i.e. abstract data objects typically represent more than one concrete object.

The abstraction/concretization approach does not take into account that there may be "orthogonal" attributes attached to the abstract value, i.e. attributes that leave the evaluation semantics of the value unchanged, but convey "extra" information. These attributes are often inserted by primitives and/or synthesized by the abstract interpreter. One example at hand is the `code` attribute used in `ac`; other possible uses of such attributes are encoding-, storage arena-, reference count- or strictness-annotations. In the following discussion, such attributes can be ignored.

The most simple kind of abstract value is a non-abstract value: an abstract value domain can be constructed "on top of" the standard value domain. This approach avoids loss of information, since all completely determined computations, i.e. all computations involving only standard values, will deliver concrete results.

The most precise abstract value domain is one that employs abstract values that represent *arbitrary sets* of standard values. The standard semantics can then be directly extended to abstract values.[3]

[3]Each function on abstract values can be reduced to its counterpart on non-standard values by enumerating the possible argument sets represented by an abstract argument list, applying the standard function to each of these lists, and merging the standard result to one abstract result.

The problem with this general abstract value type are its implementation cost: Especially when the concrete data domain contains non-finite types such as data structures and functions, arbitrary sets of these cannot be represented extensionally; even finite concrete types incur prohibitive costs (exponential in the bit-width needed to represent an instance of the concrete type). The run time of the primitive functions grows even worse, since argument combinations are to be considered.

Instead of implementing all subsets of a standard domain, most AI systems employ a *predetermined set of subsets*. One obvious choice for such subsets are the *data types*: for each type, one abstract value is introduced to model the set of all values of this type. In the current version of `ac`, "general set" abstract values are not supported[4], and all abstract values either model one concrete value or all values of a data type.

6.2.2 Domains

A *domain* is a *set* with an internal structure given by an *partial ordering relation* ("$\sqsubseteq$"), i.e. a relation that is reflexive, antisymmetric, and transitive. A domain is a *complete partial order (CPO)* if there is a smallest element $\perp$. For the purposes of abstract interpretation, the data domains of a programming language must be CPOs.[5]

The ordering relation is conventionally interpreted as "$a \sqsubseteq b$" iff "a is less defined than b". The "most defined" value (if it exists) is denoted by the symbol $\top$ ("top"), the "least defined" value is $\perp$ ("bottom").

Abstract interpretation is a process in which the least upper value for an expression is computed. If the expression is recursive (as in the case of a recursive function application), an iterative process is needed to approximate the value. Starting with a "first guess" of bottom, each iteration step will improve the approximation. Formally speaking, "improving" means that each new approximation is "at least as defined as" the previous one. The top element, if it exists, is a trivial solution for each expression (with no information content whatsoever).

Such an approximation process will be shown in the example of section 6.3 in detail.

6.2.3 Basic Domain Constructions

Any set S can be interpreted as a domain having the minimal ordering relation "=" (the equality relation is also called the *discrete ordering*).

Any domain S can be turned into the CPO $S_\perp$ by providing a distinct element $\perp_S$ and defining the ordering as

$\forall\, x, y \in S_\perp : x \sqsubseteq_{S_\perp} y \equiv x \sqsubseteq_S y \vee x = \perp_S .$

I.e., in addition to the ordering derived from S, each element is less than the newly introduced $\perp_S$. Such a $S_\perp$ definition is known as *lifting* the domain S. If S is a set, $S_\perp$ is called a *flat* domain.

Analogously, the domain $S^\top$ has the extra element $\top_S$ with

$\forall\, x, y \in S^\top : x \sqsubseteq_{S^\top} y \equiv x \sqsubseteq_S y \vee y = \top_S .$

Combining these two constructions, $S_\perp^\top$ can be defined as

$\forall\, x, y \in S_\perp^\top : x \sqsubseteq y \equiv x \sqsubseteq_S y \vee x = \perp \vee y = \top_S .$

In many abstract interpretation applications, flat domains are all that is needed. In denotational semantics for programming languages, domains with more internal structure are necessary (e.g., to represent first-order functions or recursive data structures).

[4] An earlier version provided them only for very small sets (less than four elements) and for sets that could not be generalized to a common data type more specific than $\top$.

[5] The restriction to CPOs is only necessary if recursive structures have to be given a semantics.

6.2.4 Basic Examples

There is some confusion as to the interpretation of semantic domains in terms of sets, especially where the top and bottom elements are concerned. Some of this can be cleared up by looking at some simple domains, and by considering the rules used to combine domains.

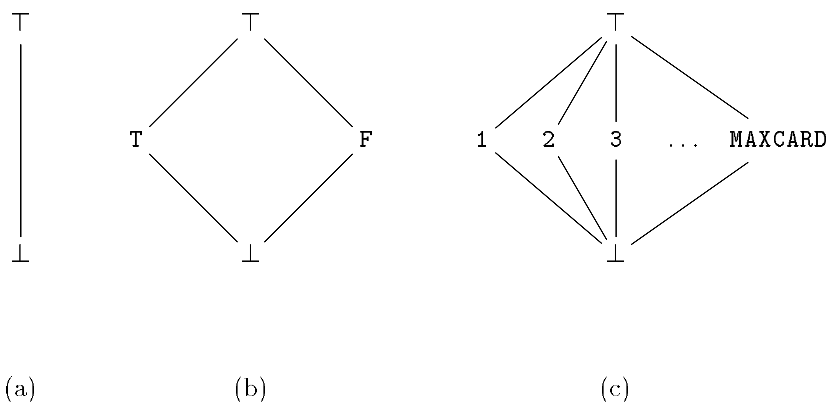

Figure 6.1: (a) 2-point domain **2**, (b) 4-point domain **Bool**, (c) (MAXCARD+2)-point-domain **Card**

The most trivial useful domain (depicted in fig. 6.1 (a)) consists of just two values, $\top$ and $\bot$. This domain is simply called **2**.[6] The domain **2** is often employed when only one relevant characteristic of each construct is of interest (e.g. in data-flow analyses such as strictness analysis).

The simplest domain that includes "real" values and the complete set of $\top$ and $\bot$ elements is the boolean value domain

$$\texttt{Bool} = \{T, F\}_{\bot}^{\top} = \{\top, T, F, \bot\}$$

(cf. fig. 6.1 (b)). In the approximation process of the abstract interpreter, these values are used as follows:

- $\bot$ - bottom, a value without any properties.
- T - this is the value of an expression that, if it terminates, evaluates to T (true).
- F - this is the value of an expression that, if it terminates, evaluates to F (false).
- $\top$ - top, a value that means "both T and F".

That is, $\top$ combines all properties of the values "below" it; in contrast, $\bot$ has none of the properties of the values "above" it. It is important to note that during a fixpoint computation, all approximation steps go "upward" (toward top), but never "downward" (toward bottom).[7]

Practically speaking, $\top$ and $\bot$ convey about the same amount of information (we don't know what the actual value is), but due to this directedness of values, a bottom-valued expression still holds the promise of finally being approximated to a distinct value (T or F). In contrast, a top-valued expression cannot be "lowered" anymore.

[6] The domains **0** (the empty set) and **1** (containing only $\bot$) are quite useless.

[7] This is really the basic idea of the fixpoint theorem: If the domain of values is of finite height (the longest $\sqsubseteq$-path between bottom and top is of finite length), and all functions are monotone, a fixpoint for any value exists and can be found in a finite number of steps.

To use a different example, we can re-interpret the `Bool` domain: instead of boolean values, T and F can be used to model even and odd numbers. Then $\bot$ is a number which is "neither even nor odd", T describes any number that is even, F describes any number that is odd, and $\top$ describes any number that is "both even and odd".

Domains can often be interpreted as (power)sets. Under such an interpretation, $\bot$ is the empty set, T the set of even numbers, F the set of odd numbers, and $\top$ the set of all numbers.

The last example is **Card**, a flat cardinal domain consisting of a set of numbers $1..MAXCARD$, together with $\top$ and $\bot$ (cf. figure 6.1 (c)). It is explicated just like the boolean domain: $\bot$ stands for the number without any properties (i.e., the empty set), each number stands for itself (i.e., a singleton set), and $\top$ stands for the number that merges the properties of all other numbers (i.e., the set of numbers). If we interpret each element as a set, the partial ordering is nothing but the subset-relation.

6.2.5 Union of Domains

Two or more domains $D_1...D_n$ can be "merged" to a domain C by defining new top and bottom elements $\top_C$ and $\bot_C$, and by defining:

$$\forall\ x\ \in\ D_i\ (i \leq \leq\ n)\ :\ x\ \sqsubseteq_C\ \top_C\ \wedge\ \bot_C\ \sqsubseteq\ x$$
$$\forall\ x, y\ \in\ D_i\ (i \leq \leq\ n)\ :\ x\ \sqsubseteq_{D_i}\ y\ \equiv\ x\ \sqsubseteq_C\ y$$

This corresponds to the union operations on sets. The definedness-relations within the C_i domains are left untouched, and values from C_i are incomparible with values from C_j (assuming $i \neq j$).

In such a hierarchic domain structure, the different top- and bottom-values appropriate a meaning with finer distinctions. Now, each $\bot_{D_i}$ conveys some "type information" which $\bot_C$ doesn't contain. Likewise, $\top_C$ has lost the "type" that each $\top_{B_i}$ contains.

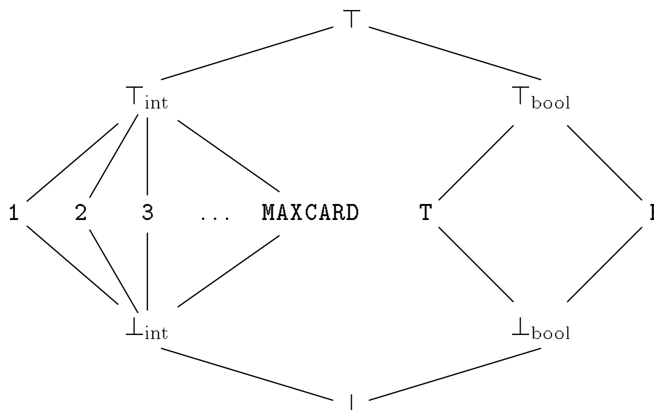

Figure 6.2: Union Domain Bool+Card

Figure 6.2 depicts the merging of the domains `Bool` and `Card`.

6.2.6 The Ideal Abstract Domain and its Subsets

Assuming a given data domain *Obj* taken from a language semantics, the *ideal abstract object domain* A^{Obj} is defined as the powerset of *Obj*, i.e. $A^{Obj} = 2^{Obj}$. Because $|Obj|$ tends to be quite large (often even infinite)

for nontrivial type systems, A^{Obj} is even more so; thus it is generally impossible to represent all possible elements of A^{Obj}, either explicitly or implicitly. A simplified abstract object domain A^{Obj}_{simpl} has to be used instead. Having this in mind, all abstract domains can be viewed as predetermined subsets of A^{Obj}.

Figure 6.3 shows an ideal abstract domain structure for the 4-value data domain `Card(2)`. The top element corresponds to the set $\{0, 1, 2, 3\}$, the bottom elemeent to the empty set. Elements of this domain could be represented by 4-bit bitvectors.

Note that there are no distinctions made at the level "below" the one-element abstract values. While it is possible to mirror the structure "above" the one-element level, there is no need for it. The two basic operations that create abstract values are

- *injection* of data values (this creates one-element abstract values), and
- *finding the least upper bound* of two or more values (this "merges" the arguments into a value "higher up" in the hierarchy).

The bottom value itself is sometimes injected as a special, unique constant that initializes approximation chains (and serves as a default value for erroneous computations).

For larger data domains, the ideal abstract domain cannot be efficiently represented: In the case of 16-bit numbers, each abstract value would be represented a 8kbyte bitvector. Even worse, arithmetic primitives would take a time at best proportional to the number of bits, at worst proportional to the squared number of bits (in the case of binary operations).

Obviously, non-trivial data domains call for simplified abstract domains.

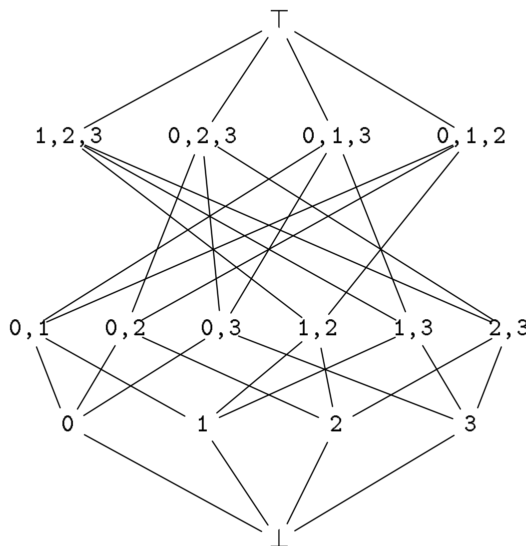

Figure 6.3: Ideal Abstract Domain $2^{Card(2)}$

One practical approximation employs *ranges*:

$A^{Int}_{range} = \{[a, b] | a, b \in \ Int \ \wedge \ a \leq b\} \ \cup \ \{\perp, \top\}$.

A range object $[a, b]$ is interpreted as the set $\{x | x \geq \ a \ \wedge \ x \leq \ b\}$. An integer n is represented as $[n, n]$. $\perp$ and $\top$ need not be extra elements: $\perp$ can be interpreted as the (unique) empty interval, and $\top$ as the interval $[MinInt, MaxInt]$.

Likewise, an abstract representation of the IEEE-754 real numbers

$$Float = \{m * 2^n \mid m \in Int_{mantwidth},\ n \in Int_{expwidth}\} \cup \{+\infty, -\infty, -0, NaN\}$$

could be made using values from the domain

$$A^{Float}_{range} = \{[a..b] \mid a, b \in Float \wedge a \leq b\} \cup \{\perp, \top\}.$$

This direct range domain has some small problems, namely ranges involving the "unreal" numbers $\{+\infty$, $-\infty$, -0, $NaN\}$. It might be better to split up the `Float` domain and define

$$Float' = \{m * 2^n \mid m \in Int_{mantwidth},\ n \in Int_{expwidth}\}$$
$$A^{Float'}_{range} = \{[a..b] \mid a, b \in Float' \wedge a \leq b\} \cup \{+\infty, -\infty, -0, NaN\} \cup \{\perp, \top\}$$

There is a pitfall in designing abstract domains, especially when numbers are involved: one could argue that there are values of special interest that ought to be handled separately, e.g. the set

$$Frac = \{\tfrac{n}{m} \mid n, m \in Int\}$$

or even things like

$$Frac_e = \{\tfrac{n}{m} e \mid n, m \in Int\}$$
$$Frac_\pi = \{\tfrac{n}{m} \pi \mid n, m \in Int\}$$

By adding such abstract values, though, one can easily create situations in which the abstract interpretation will show a different behaviour than the standard interpretation due to differences of rounding and precision.[8]

Using A^{Obj}_{range}s instead of A^{Obj}s results in a loss of precision, but one can usually adapt the approximation to the needs of the application. In the context of abstract interpretation, such a loss takes the form that the actual result is less precise than the optimal result, i.e. $res_{optimum} \sqsubseteq res_{actual}$.

6.2.7 Size of Domains

Many applications, most notably strictness analysis and binding time analysis (mentioned in 5.6), can achieve significant results using only the most primitive of domains: the 2-point domain **2**.

One motivation behind keeping domains small is the expense incurred by complex abstract domains: implementing the primitive functions becomes an arduous and error-prone task, and the time needed to compute fixpoints may grow exponentially in the "depth" of the domain.[9] "Depth" here means the length of the longest "$\sqsubset$"-chain of abstract values. For example, the A^{Int16}_{range}-domain of ranges over 16-bit-values has a depth of 2^{16}, since

$$[0, 0] \sqsubset [-1, 0] \sqsubset [-2, 0] \sqsubset \ldots \sqsubset [-2^{15}, 0] \sqsubset [-2^{15}, 1] \sqsubset \ldots \sqsubset [-2^{15}, 2^{15} - 1]$$

[8] Similar problems occur whenever floating-point expressions are reordered or otherwise "optimized" at compile time. Only a few of the standard algebraic transformations are valid on floating-point numbers.

[9] Most functions that occur in practical programs do not behave exponentially under iterative approximation. On the other hand, even linear growth can be quite sufficient to make AI impractical.

The most simple abstract value domain in which sensible computations can be performed is probably that of *basic types*: for each distinct type in the domain, one abstract value is introduced. For the following examples, we could define

$$
\begin{aligned}
&Bool = \{True,\ False\}\\
&Int = \{\ MININT\ \ldots,\ -1, 0, 1,\ \ldots\ MAXINT\ \}\\
&Obj = Bool + Int\\
&A^{Obj}_{basic} = Obj \cup \{\texttt{anInt}, \texttt{aBool}\} \cup \{\bot, \top\}
\end{aligned}
$$

Here, the depth of the domain is 4:

$$
\begin{aligned}
&\forall\ x \in Obj_{int} : \bot \sqsubset x \sqsubset \texttt{anInt} \sqsubset \top\\
&\forall\ x \in Obj_{bool} : \bot \sqsubset x \sqsubset \texttt{aBool} \sqsubset \top
\end{aligned}
$$

A more precise approximation could be one oriented along *machine types*:

$$A^{Obj}_{machine} = Obj \cup \{\texttt{Bit}, \texttt{Byte}, \texttt{Word}, \texttt{Long}, \bot, \top\}$$

with `Bit` $\equiv$ `aBool` and a depth of 7:

$$\forall\ x \in \{0, 1\} : \bot \sqsubset x \sqsubset Bit \sqsubset Byte \sqsubset Word \sqsubset Long \sqsubset \top$$

A good compromise between efficiency and precision, and the one employed by the abstract interpreter in `ac`, is a domain built around the bit-widths of numeric values.

$$A^{Obj}_{width} = Obj \cup \{Int(n) | n \in 1 \ldots maxwidth\} \cup \{Card(n) | n \in 1 \ldots maxwidth\} \cup \{\texttt{aBool}, \bot, \top\}$$

6.2.8 Sources of Abstract Values

Where do the abstract values come from? It is certainly impossible for a source program to contain abstract literals, and all non-side-effect operations taking concrete arguments deliver concrete results.[10]

There are three sources of abstract values:

- Side-effecting operations, especially input functions. In ALDiSP, a primitive function like `read`, applied to, e.g., a `Cardinal(16)`-port, will return an abstract value representing all possible unsigned 16-bit values.[11]
- Type annotations: If a function is annotated with explicit argument types, that information can be used to synthesize abstract values. This approach is used when abstract interpretation is done on a function-by-function basis, i.e. when not all call points are known in advance. In a framework that supports separate compilation, an abstract interpreter will have to use such a strategy for all functions that are "exported".

[10] With the possible exception of non-deterministic functions like `random()` or `choose()`. Since no such function is part of the ALDiSP definition, this can be ignored.

[11] For the purpose of equality tests, such a value should be annotated with a "source" attribute, e.g. "the value of input port x at time t". In `ac`, this is implied by the `atom/code` attribute (cf. section 8.1).

- Manually inserted values: Some experimental partial evaluators are targeted at manual intervention; a lot of the published work belongs to this class.

In `ac`, input primitives are the sole initial source of abstract values. The program is abstractly interpreted as a whole; libraries are "linked" by textual inclusion of their source, so the possible call points for all functions are known at compile time. The abstract interpreter is not visible to the programmer[12], therefore manual intervention is not needed.

6.3 An Example

Consider the well-known example function

```
func fak(n) =
   if n==0 then 1
           else n*fak(n-1)
   end
```

and how the AI may process it. For a start we assume that the AI works with an abstract domain of booleans and integer intervals:

$$A_{example} = \{[a,b] \mid a \leq b \wedge a,b \in \mathit{Int}\} \cup \{\mathit{false}, \mathit{true}, \mathit{aBool}\} \cup \{\bot, \top\}$$

Let `fak` be called with the argument `[0,100]`, i.e. some unknown value between 0 and 100, inclusive. Evaluation starts off with the comparison `[0,100]==0`, which reduces to `aBool`. What is the `if` to do?

6.3.1 Abstract Conditionals

When the test of a conditional expression does not evaluate to a concrete result, all possible branches covered by the abstract selector must be evaluated, and the results must be *merged.*[13] Abstract values are merged by applying the concretization function to them, building the union of the resulting sets, and finding the "best approximation" from the domain of abstract values, i.e. that abstract value which maps to the smallest set of standard values of which that union is a subset. For example, merging $[1,2]$ and $[10,11]$ would result in $[1,11]$. Other values, e.g. $[-12,72]$ or $\top$, would be as "correct", but needlessly imprecise. In the context of the partial order of the domain, the merge operation corresponds to the *least upper bound* (lub) on domains, i.e. the least defined value from which $[1,2]$ and $[10,11]$ are specializations. The least upper bound is denoted by the infix operator "$\sqcup$".

While the basic idea behind the treatment of conditionals is obvious, there are a number of practical problems.

In the example, both branches of the `if` must be evaluated. `1` evaluates to the singleton interval $[1,1]$, and `n-1` evaluates to $[-1,99]$.

Here, the first problem emerges: In real execution, `n-1` can never evaluate to `-1`; this is consequence of the test `n==0` being `false` in the `else` path. The abstract interpreter has ignored this precondition. What are the consequences of this "loss of information" ?

`fak` will be called recursively with arguments

$[-1,99],[-2,98],...,[-100,0],[-101,-1],[-102,-2],...$

[12] It might, however, be visible to the library writer. The library can communicate with the compiler by employing primitive functions that directly manipulate the abstract interpreter heuristics, or that depend upon the compilation state.

[13] While determining the set of possible branches is trivial in "`Lcond`" expressions, it becomes interesting in the case of "`Lselect`" expressions and numeric range abstract values.

When evaluating $[-101,-1]$==0 (and all following comparisons), the result will be `false`: the AI will not terminate on `fak(`$[0,100]$`])`, while the standard interpretation of `fak` terminates after a maximum of 100 recursive calls.

This is one consequence of information loss: terminating loops may become non-terminating.

What happens if the abstract interpreter is smart enough to "split" the abstract value of `n` in the different execution paths of the `if`? The initial value of $[0,100]$ can be split into $[0,0]$ and $[1,100]$ so that `fak` is called with $[0,99], ..., [0,1], [0,0]$ and safely terminates. Now the merging starts: because for all `n`, $[1,n]$ merged with $[1,1]$ is $[1,n]$, the merges with $[1,1]$ can be ignored. The result of the call chain will be $[1,1]$*$[1,2]$*$[1,3]$*...*$[1,100]$, i.e, $[1!, 100!]$. This is, indeed, a perfect result – not quite as good as possible, because the range representation denotes $9.33262 * 10^{157}$ numbers while there are but 100 possible "real" results, but the best we can hope for within the context of the chosen abstract domain.

A "splitting" mechanism would thus be a very good thing to incorporate in the AI; its implementation, though, is – in general – impossible. There are two reasons for this: first, the inverse function of a predicate has to be computed. If we allow arbitrary recursive predicates and infinite domains, this is impossible. If we restrict ourselves to finite domains, the inverses can be computed, but not necessarily represented: an ideal abstract domain would be needed for this. In practice, this second problem is far worse than the first one: at least in DSP programming, conditions tend to be simple comparisons and non-recursive predicates. But even the most trivial predicate (comparison against a literal) can easily create splittings that cannot be represented by an abstract domain of consecutive subranges.

It can therefore not be assumed that a splitting device exists. Since an abstract interpreter will always need some safety mechanism to stop runaway loops – after all, a source program may contain a "genuine" nonterminating function, and an AI should terminate on any input –, we assume that some kind of *loop detector* guards all function calls.

6.3.2 Loop Detection – Finding Fixed Points

In ALDiSP and most other functional languages, loops can only occur through recursive function calls. It is therefore guaranteed that any looping path of control has to pass at least one function application. A loop detector installed at the "apply" point can therefore preempt all nonterminating behaviour, thus guaranteeing the termination of the abstract interpreter.

We assume that the loop detector decides that `fak(`$[-1,99]$`)` is sufficiently "similar" to the still unfinished evaluation of `fak(`$[0,100]$`)` to call it a loop.

The abstract interpreter therefore aborts the further evaluation of `fak(`$[-1,99]$`)` and returns at once with a "faked" return value: $\perp$, the empty result set. $\perp$ is chosen as the initial value, since it is the "weakest" or "least defined" approximation to the real, as yet unknown, result.

Evaluation continues with the `*` operator. $[0,100] * \perp$ is defined as $\perp$. This bottom-preserving behaviour is one of the foundations of the semantic machinery; it can also be derived from the definition of abstract values via sets.

The evaluation of both arms of the `if` is now finished, the results are combined: $\perp \sqcup [1,1] = [1,1]$, since $\perp$ is the neutral element for $\sqcup$.

Thus ends the first attempt to determine the value of `fak(`$[0,100]$`)`. The result, $[1,1]$, is somewhat disappointing. But the process can be iterated! The crucial step, returning $\perp$ as the value of `fak(`$[-1,99]$`)`, was only a first approximation – now, $[1,1]$ can be returned (since we think that $[0,100]$ is "similar" to $[-1,99]$, we assume that `fak(`$[0,100]$`)` is also similar to `fak(`$[-1,99]$`)`); this is still not the "correct" result, but better than $\perp$. $[1,1]$*$[1,100]$ is $[1,100]$. The approximation continues: the next cycle delivers $[1,100]$ as result of `fak(`$[-1,99]$`)`. $[1,100]$*$[1,100]$ is $[1,10000]$. Now we see why the "depth" of the abstract value

domain must be finite: if there is no upper bound to the size of an integer (and thus to the depth of the domain), the approximation will not terminate. The domain can be restricted to finite depth by a number of means. As far as numeric representations are concerned, the base domains are finite in practice, so the introduction of an upper limit (`MaxInt`) makes sense. Any (left or right side of an) abstract result exceeding this is transformed to (an interval containing) `MaxInt`. Primitive operations are closed over this system, e.g. [1,`MaxInt`]`*`[1,`MaxInt`] is [1,`MaxInt`]. The `fak` example would then return an abstract result of [1,`MaxInt`].

The result [1,`MaxInt`] is a *fixed point* of the `fak` function: assuming

$$\forall\ x\ \in\ Int_{>0} :\ f(x)\ :=\ [1,\ MaxInt]$$

it can be verified that

$$\forall\ x\ \in\ Int_{>0} :\ (if\ x = 0\ then\ 1\ else\ n * f(x-1)\ end)\ \equiv\ [1, MaxInt]$$

If there are only finite approximation chains, approximation by iteration is guaranteed to find a fixed point in finite time.

6.3.3 Speed

The mere existence of a fixed point does not help very much if the approximation takes a long time because it consists of too many steps – say, $[1,1],[1,2],[1,3],\ldots,[1,\texttt{MaxInt}]$. While it is encouraging to know that the partial evaluator will eventually stop, it is pragmatically unsatisfactory if there is no small upper bound to the number of iterations it takes.

The computation of fixed points is of a complexity exponential in the depth of the abstract domain.[14]. Methods of speeding up the iteration process are described in the literature. It is usually quite easy to find a safe approximation to the fixed point; if the domain contains a top element ($\top$), this element *is* such an approximation. But of course, no knowledge at all is gained by this approximation: $\top$ models "any result".

A practical way to compute the fixed point of a function — indeed, the approach taken by `ac` — is to switch to a "coarse" abstraction when too much time has been spent on a call. The "coarsest useful" abstraction level is that of "implementation types", i.e., the types that correspond to basic machine representation categories.

The heuristic that is implemented in current interpreter (to be more specific, in the $\texttt{similar}_{\text{group}}$ function; it defines the similarity measure and generalization of groups of argument lists) works as follows: if a certain depth of the call stack is exceeded, the argument lists are generalized to their basic types. This threshold is usually set to values between 3 and 20. If all arguments are already at the level of basic types, or if a second threshold (usually set to 100) is exceeded, the arguments are generalized to $\top$. Thus termination is enforced.

6.3.4 Generalizing the Argument Lists

The example, as explained above, did include some hand-waving: what is the legitimation of using the result of `fak`$([0,100])$ as an approximation for the result of `fak`$([-1,99])$? There is none, because a crucial step has been left out of the example, namely *argument generalization.* The loop detector believes to have found a loop when there is a call to a function f with an argument list $(a_1,\ldots,a_n)$, when another call to f with a

[14] The exponential worst-case complexity is for first-order functions [91] [66]; the situation for higher-order functions is even worse.

similar argument list $(a'_1, \ldots, a'_n)$ is already on the call stack. The approximation process returns an answer for

$$f(a_1 \sqcup a'_1, \ldots, a_n \sqcup a'_n)$$

i.e. for both calls, the one at hand and the one in the call stack. Since it is always correct to replace an abstract value a by an abstract value a' if $a \sqsubseteq a'$, it is correct to re-run the approximation process. On each run, the argument domain may become less specific (since the initial argument list is generalized each time), but a fixed point will eventually be found.

6.3.5 Conclusion of the Example

These two mechanisms, handling of conditionals and finding fixed points through iterative approximation and generalization, are sufficient to convert a standard semantics interpreter for a first-order language into an abstract interpreter. Sections 6.5.1ff. will show special problems that do not emerge in a first-order, statically-bound function like `fak`.[15]

6.4 Implementing the Loop Detector

The presence of a loop detection scheme is probably the most important difference between a simple and an abstract interpreter. As already mentioned, such a scheme is based on a function application cache.[16]

The call cache keeps track of all applications of closures. Only closures have to be considered, since applications of primitive functions and arrays cannot give rise to looping behaviour, and overloaded functions are eventually reduced to closures. For each application, the call cache has to know the argument list, the current state and the exception environment. State and exception environment can be handled as "implicit arguments" to a function call.

Each call in the cache is marked as either "open", "closed", or "approximated". The call cache can be described as a partial function

$\texttt{CallCache} : \mathit{func} \times \mathit{args} \times \mathit{context} \rightarrow \mathit{status}$

with

$\mathit{status} = \{\texttt{undefined}, \texttt{open}, \texttt{closed}(\mathit{result}), \texttt{approx}(\mathit{result})\}$

The `undefined` status is only needed to make the `CallCache` function total.

When a function is first called with a given list of arguments, it is marked "open". The call is "closed" upon return from the function; the return value can, but need not be, memoized. A loop is detected whenever a function is called with a set of arguments for which it is marked open. The call cache can either be implemented as a global data structure maintained by side-effects, or in a more functional manner by keeping track of the "current call stack" in the interpreter. The latter approach, while being more elegant and general, has some drawbacks:

---

[15] In typical DSP programs, nearly all of the program code consists of first-order function definitions, and all but an insignificant amount of the actual run time is spent in these functions. The example can therefore be treated as representative.

[16] It is interesting to ponder a design that lives without such a cache. One viable alternative is to integrate the fixpoint operator (cf. sec. 3.4.4) into the loop detection scheme. The fixpoint operator is usually considered as a device that "ties a knot" into the environment structure. It can also be understood and implemented as a function that generates new incarnations of a recursive function by need. A fixpoint function that knows about the argument and result types and keeps track of its the previously generated sibling incarnations can then replace the actual code of the function body with a call to a matching, already generated incarnation. There is still a small residue of the call stack, but it can be localized in each function's fixpoint operator incarnation. An implementation with known and fixed strategies and heuristics might employ such a integrated approach to semantics, loop detection, and code generation. For the present implementation, which is rather experimental and open with respect to new heuristics and strategies, this approach was considered too complex and not "modular" enough.

- each interpreter function has to be extended to handle an extra parameter (but that can be incorporated into the catch-all "`Context`")
- there is the danger of a "space leak": the call cache is easily the biggest data structure constructed during the AI. If it is kept functional, old copies of it might be kept around far beyond their needed lifetime. Even with call cache implemented by side-effects, the AI consumes vast amounts of space (single runs easily allocate 50 Mbyte); a functional cache could only worsen allocation behaviour.

Since most of the contents of the call cache are needed anyway by the code reconstruction phase (cf. chapter 8), the cache can as well be held as a global data structure.

6.4.1 Naïve Loop Detection Schemes

First a "naïve" loop detection scheme is presented. It is capable of handling *direct* loops correctly and efficiently.

The call cache is assumed to be globally accessible and modifiable with the `:=` operator. The iteration control is given as an extension to the standard semantics. The function $\texttt{apply}_{\text{closure}}$ of the standard semantics is renamed to $\texttt{apply}_{\text{closure/standard}}$, and a new $\texttt{apply}_{\text{closure}}$ is defined to be

```
apply_closure(f,args,C) =
  case CallCache(f,args,C)
    of open      => CallCache(f,args,C) := approx(⊥);
                    result(⊥,S)
     | approx(r)
     | closed(r) => r
     | undefined => CallCache(f,args,C) := open;
                    apply_approx(f,args,c)
apply_approx(f,args,C) =
  r = apply_closure/standard(f,args,C);
  case CallCache(f,args,C)
    of open => CallCache(f,args,C) := closed(r);
               r
     | approx(a), a = r  => CallCache(f,args,C) := closed(r);
                            a
     | approx(a), a ≠ r => CallCache(f,args,C) := approx(r ⊔ a);
                            apply_approx(f,args,C)
```

When a looping function is called the first time with a new set of arguments, its call cache entry is undefined. A new entry is marked as `open`, and the standard application is computed. If a second call with the same arguments happens during this "open" state, an initial approximation ($\perp$) is entered into the cache, and the evaluation aborts with this value. All succeeding looping calls will directly return the approximation value, without changing the cache.

Upon return from $\texttt{apply}_{\text{closure/standard}}$, the call cache status is inspected. If there is no change (i.e., the entry is still marked "`open`"), no loop has been detected during the application, and no approximation is necessary: the cache can be closed, the result memoized and returned.

If there is an `approx` entry, that entry has to be compared with the actual result (`r`). If they are identical, the approximation is *stable*, and the entry can again be closed. If the approximation is not stable, the new approximation is merged with the old one, and $\texttt{apply}_{\text{approx}}$ is called again. No generalization of the arguments lists (as described in section 6.3.4) is necessary, since a loop is only detected upon total argument matches.

6.4.2 *Problems with the Naïve Scheme*

For several reasons, the recursion approximation scheme presented in the previous section is unable to cope with all programs.

6.4.2.1 *Not Enough Generalization*

The biggest problem is the treatment of "near misses", i.e. small variations in the argument lists. Take, for example, the "`FindFirst`" function, which finds the index of the first occurrence of a value in a vector:

```
func FindFirst(obj:Sample,vec:Vector(Sample)) = \\ this aldisp function
  let                                          \\ will really tax a
    func loop(i:Card) =                        \\ naive loop scheme!
      if i >= sizeOf(vec) then -1
      [] vec(i) == obj then i
                        else loop(i+1)
      end
  in
    loop(0)
  end
```

Assume that the size of the vector is not known at compile time, i.e. that it is input dependent; then `i >= sizeOf(vec)` will evaluate to "*aBool*" every time. Under the naïve scheme, the abstract interpretation of the `loop` function will never terminate, since `loop` is called with a different argument (0,1,2,...) each time![17]

If the vector size is statically known, the naïve scheme will lead to a total unrolling.[18]

A simple solution to this problem consists of performing a "generalized comparison", i.e. to replace the

```
case CallCache(f,args,C)
  of ...
```

in $\texttt{apply}_{\text{closure}}$ with a

```
if ∃ args': similar(args',args) ∧ CallCache(f,args',C) = ... then ...
```

The similarity relation can be defined in terms of a *generalization function*:

```
similar(args,args') =
  |args| = |args'| ∧ ∀ i ∈ 1..|args|: generalize(args_i) = generalize(args'_i)
```

One obvious choice for a generalization function is the "base type" function. Two argument lists would then be considered similar when their corresponding elements are of the same base type.

6.4.2.2 *Too much Generalization*

But it is not enough to replace the equality test with a similarity relation: after all, we want *some* level of loop unrolling in the abstract interpreter! Consider the function

[17] The interpretation will not even stop at 2^{16} or 2^{32}, since the literal "`0`" is not typed; the counting will therefore be done in infinite-precision arithmetic.

[18] This proves to be catastrophical in practice, since the abstract interpreter will most likely exceed the resource limitations of the system it runs on; if not in time, then in space. In `ac`, unrolling up to 4 or 8 iterations seems to be reasonable.

```
func pow(x,n) =
    if n == 0 then 1
    [] n == 1 then x
    else
      let
        pow_xx = pow(x*x,n div 2)
      in
        if n mod 2 == 0 then pow_xx
                        else pow_xx * x
        end
      end
    end
```

A function like `pow` is a prime target for inlining, since the second argument is often known at compile time, and unrolling removes most of the "expensive" code (division and modulo, comparison and conditional).

A call to `pow` with known second argument, e.g. `pow(x,5)`, should later expand to CF code (cf. chapter 8)

```
[ pow_xx/1 = x * x
  pow_xx/2 = pow_xx/1 * pow_xx/1
  pow_xx/3 = pox_xx/2 * x
| pow_xx/3 ]
```

but under a similarity relation based on generalization to base types, no such unrolling would occur. On most target architectures, the generalized code will be much more expensive than the unrolled (i.e., specialized) version. To facilitate unrolling, the loop detector should only be triggered when some depth of recursion is exceeded. To accommodate this, the $\texttt{apply}_{\text{closure}}$ function has to be extended to handle groups of arguments lists:

$$\texttt{if } \exists\ \texttt{args}_1,\ldots,\texttt{args}_n\texttt{: } \forall\ \texttt{i} \in \texttt{1..n: CallCache(f,args}_i\texttt{,C) = open}$$
$$\wedge\ \texttt{similar}_{\text{group}}\texttt{(\{args,args}_1,\ldots,\texttt{args}_i\texttt{\}) > match_threshold then} \ldots$$

$\texttt{similar}_{\text{group}}$ can be built as an extension of the `similar` predicate; it should take into account the size of the set to be compared. On a large set, the pressure to find a loop is bigger, and a more general similarity predicate can be used than on a small set. As a limit, if an exact match is found, a set of size 2 is large enough for the similarity to become 1.0. If the set is beyond a certain size (say, 100), strong generalization (up to $\top$) can be allowed. While no realistic program will have a need for $\top$-generalization (i.e., generalization beyond the basic-type level), its presence guarantees that AI eventually finds a loop in all recursive functions.[19]

6.4.2.3 Mutual Recursion

What happens if two or more functions call each other recursively? Consider the case of simple mutual recursion, as exemplified by the `even/odd` example:

```
func even(n)= if n == 0 then true
                        else odd(n-1)
              end
func odd(n) = if n == 0 then false
                        else even(n-1)
              end
```

When `even` is called with an abstract argument, the ensuing call stack will consist of alternating applications of `even` and `odd`. The fixed point finding mechanism may be triggered at the k-th call to `even`, for some

[19] This is *not* a guarantee that the AI as a whole terminates, since the set of possible functions is not finite. By generating new functions (i.e., closures) on the fly, nontermination is still possible. A second source of nontermination lurks in the implicit de-referencing, which can cause loops not caused by recursive function applications.

k depending upon the internal structure of the abstract value bound to `n` and the heuristics employed by the similarity predicate. In each iteration of the fixed point finder, `odd` will be called, and a value will be returned. Since that value will only be an approximation, it would be erroneous to store it in a "closed" call cache entry (CCE) for `odd`; if this were done, following iterations might be aborted before encountering the new approximation. It is therefore necessary to "flush" the call cache of all entries that were constructed during an approximation iteration.

After the iterations have found a fixed point, the CCE for `even` will be closed; its `code` attribute will refer to a CCE for `odd`, which is also closed, and which is defined in terms of the (recursive) CCE of `even`. `odd` might even be inlined, since it does not refer to itself, but only to `even`.

6.4.2.4 Multivariant Specialization

Some functions can have two or more *specializations*. A specialization is characterized by an argument set for which more efficient code can be generated than for other argument sets. Consider this transformation of the `even/odd` example into one function with a boolean parameter[20]

```
func eo(n,eflag) =
  if eflag then if n == 0 then true else eo(n-1,false) end
           else if n == 0 then false else eo(n-1,true) end
func even'(n) = eo(n,true)
func odd'(n) = eo(n,false)
```

An application of `even'` or `odd'` to an unknown `n` will result in recursive calls that can be specialized in two distinct ways: since `eflag` is always known, code has to be generated for only one of the forks of the `if eflag` condition. That is, specialization should generate the original even/odd function pair, since the `eflag` parameter can then be dropped. To facilitate such multivariant specialization, the generalization strategy must avoid merging the `eflag` position of a generalized argument list to `aBool`. In an abstract domain that consists only of literals and base type values, this would be no problem, since every third recursive call to `eo` would be an exact match in any case; but in a generalization that employs a more detailed abstract value hierarchy (e.g. interval arithmetic), a match might occur only after a recursion depth > 3.

Since the similarity predicate employs the generalization function anyway, it makes sense that it also generates the appropriate generalized argument lists. $\texttt{similar}_{\text{group}}$ is then of type

```
similar_group : (Argument list) Set -> Similarity * (Argument list)
Similarity = [0...1]
```

The similarity relation might then detect that the elements of a *subgroup* of the argument lists are more similar to each other than to the other argument lists on the stack is, and return generalized arguments for that subgroup only.

6.4.2.5 Restarting Approximations

In the naïve loop detector described in section 6.4.1, the first iteration started on the "way back" from the point where the loop was detected. This is not possible in a generalized scheme, since an arbitrary number of recursion steps, some of them unrelated (e.g. through multivariant specialization) to the looping function, may reside on the call stack. The generalized scheme therefore does a *restart* from the beginning each time

[20] This example is obviously inefficient. When the loop detector is applied to the optimized variation `if n==0 then eflag else eo(n-1,not eflag) end`, multivariant specialization does not gain anything. The example was chosen for its simplicity only. As for most "optimizing" program transformations, there is no guaranteed win in multivariant specialization. In "real-world" cases, candidates for multivariant specialization can often be found by looking for array index parameters used with a constant stride or offset.

a loop is detected. Also, all open calls that are "between" the entry point of the loop and the point where the loop was detected have to be removed, since they would only pollute the cache, and misguide the loop approximation of other functions.

How can the "restart" be implemented? One possibility is to store the *continuation* of each application in the `open` call cache entry. Restarting can then directly be modelled and implemented as call to the "restart continuation". This approach has two problems:

- Memory consumption: each `open` entry would contain the whole state of the abstract interpreter at the point of $\texttt{apply}_{\text{closure}}$. Garbage collection (of the underlying SML run-time system that supports the data structures of the abstract interpreter implemented for `ac`) would be unable to remove such old interpreter state. Manual removal of "dead" `open` entries would become necessary, since in many ALDiSP programs, functions change state between loop iterations. For such functions, `apply` loop detection is subsumed by the state loop detection implemented in the scheduler.
- How is the "current continuation" accessed? The whole semantics would have to be re-written if the $\texttt{apply}_{\text{closure}}$ needs an explicit continuation parameter. Implicit continuations are not supported by the semantics. Even if they were: SML/NJ has a `call/cc` primitive, but it is not part of the SML standard, and would render the compiler non-portable.

To circumvent these problems, `ac` resorts to a strategy dubbed *in-band signalling*, since the "control information" is encoded within the "data stream": upon encountering a loop, computation is aborted and a *loop exception* is returned. Each CCE is now *uniquely tagged* and the loop exception contains the tag of the CCE from which the loop *originates*. The originating CCE is determined by looking for the "earliest" entry among those that are found to be similar. To make this operation possible, the CCE tags are time-stamped. Each `apply` function, upon recognizing the loop exception value, either invalidates its CCE (if the tag doesn't match), or restarts the computation (if it does match).

6.4.3 Generalizing Loop Detection

The following loop detection semantics contains all the changes described in the previous sections. The similarity relation is assumed to be specified in terms of the generalization function. No assumptions are made about the generalization function.

The call cache is now indexed by tags; it is of type

$$\texttt{CallCache} : \mathit{Tag} \rightarrow (\mathit{Function} \times \mathit{Arguments} \times \mathit{Context} \times \mathit{Status}) \cup \{\texttt{undefined}\}$$
$$\mathit{Status} = \{\texttt{open}, \texttt{closed}(\mathit{Result}), \texttt{approx}(\mathit{Result})\}$$

The $\texttt{apply}_{\text{closure/standard}}$ function is now supplied as an extra parameter to the loop detector, which has been renamed `call`. The new definition of $\texttt{apply}_{\text{closure}}$ function (which models the actual implementation used in the compiler) is:

$\texttt{apply(f,args,C) = call(f,args,C,apply}_{\text{closure/standard}}\texttt{)}$

By providing the actual "primitive" application function as a parameter to the `call` and $\texttt{loop}_{\text{approx}}$ functions, the loop detector can be specified and implemented within a separate module. This enhances modularity and simplifies testing, since different "strategies" can be attached to the same abstract semantics.

```
call(f,args,C,applyfn) =
  if ∃ tag_approx : CallCache(tag_approx) = (f,args_approx,C,approx(r))
     ∧ similar(args_approx, args)
  then
    CallCache(tag_approx) := (f,args_approx ⊔ args,C,approx(r));
    r
  else
    if match_degree < match_threshold then
      tag_new = new_tag();
      CallCache(tag_new) := (f,args,C,open);
      res = applyfn(f,args,C)
      case res
        of exception_loop(tag,args_generalized) =>
          if tag = tag_new then
             loop_approx(applyfn,f,tag,C,args_generalized,⊥)
          else
             CallCache(tag_new) = undefined;
             res
        else
          CallCache(tag_new) := (f,args,C,closed(res));
          res
    else
      exception_loop(Min(T_bestmatch),args_generalized)
  where
    (match_degree, args_generalized) = similar_group{args, args_s0,...,args_sk}
    T_bestmatch = {t_b0,...,t_bj} = Max_simset(2^T)
    T={tag_1,...,tag_n}
    where
      ∀ i∈ 1..n: CallCache(tag_i) = (f,args_i,C,open)
      simset({t_s0,...,t_sk}) = sim_degree
        where
          (sim_degree, args_generalized) = similar_group{args,args_s0,...,args_sk}
```

The `call` function first looks for call cache entries that match the function to be applied and the current context (state and exception environment). If there is such an entry, and it contains an "`approx(`*result*`)`" value, evaluation aborts with the stored approximation *result*. The approx value is compared to the current argument list using the simple similarity relation. If the application is aborted, the argument list of the approximation is generalized by being merged with the current argument list (cf. section 6.3.4).

If no approximation is in progress, the $\mathtt{match}_{\mathrm{degree}}$ is computed. First, the set `T` of all (tags of) call cache entries that might be part of a loop is determined. This set consists of all open applications of the current function within the current context. For each subset of `T`, i.e. for each element of 2^{T}, the $\mathtt{sim}_{\mathrm{degree}}$ is determined; the details are deferred to the $\mathtt{similar}_{\mathrm{group}}$ function. The subset with the best (=highest) degree of similarity is chosen as $\mathtt{T}_{\mathrm{bestmatch}}$. If this maximum similarity exceeds the $\mathtt{match}_{\mathrm{threshold}}$, a loop is detected, and the $\mathtt{exception}_{\mathrm{loop}}$ is returned. This exception contains two values, the generalized argument list of $\mathtt{args}_{\mathrm{generalized}}$ and the CCE tag that corresponds to the loop entry point. To determine this tag, the CCE tags have to be ordered in "time", i.e. in order of their generation. The "earliest" tag in $\mathtt{T}_{\mathrm{bestmatch}}$ corresponds to the entry point.

If no approximation is in progress and no loop is detected, a new CCE tag is allocated, the current arguments and context are entered into the call cache, and the function is applied using the `applyfn`. If the result is *not* $\mathtt{exception}_{\mathrm{loop}}$, the call cache entry can be closed.

If the application's result is of the form $\mathtt{exception}_{\mathrm{loop}}(\mathtt{tag}_{\mathrm{entry}},\mathtt{args})$, a loop has been detected further

"down" the call cache, and $\texttt{tag}_{\text{entry}}$ denotes the (CCE tag of the) loop entry point. If $\texttt{tag}_{\text{entry}}$ does not match the current tag, the current entry must be discarded, and the exception handed up as result. If the tag matches, the iterative approximation process as specified by $\texttt{loop}_{\text{approx}}$ starts:

```
loop_approx(applyfn,f,tag,C,args_current,res_current) =
  let
    ∀ t ∈ Tags(CallCache) :
       if t > tag then CallCache(t) := undefined;
    CallCache(tag) := (f,args_current,C,approx(res_current));
    res_new = applyfn(f,args_current,C)
    (f,args_new,C,approx(res_current)) = CallCache(tag)
  in
    if args_current =_value args_new
      ∧ res_current =_value res_new
      then
        CallCache(tag) := (f,args,C,closed(res_new)) ;
        res_new
      else
        loop_approx(applyfn,f,tag,C,args_new,res_new)
```

$\texttt{loop}_{\text{approx}}$ takes five paramters:

- the semantic function that implements the application (`applyfn`),
- the LF function for which an approximation is to be found (`f`),
- the `tag` that identifies the call cache entry which is the "root" of the approximation process (i.e., the first application of the function with the generalizable argument set),
- the current evaluation context (`C`),
- the argument list that was used to compute the last approximation ($\texttt{args}_{\text{current}}$), and
- the result value that was generated by the last approximation ($\texttt{res}_{\text{current}}$).

The behaviour of the approximation loop is determined by the convergence of argument list and result approximation during the iterations. If the results of two successive iterations have the same semantic value, and if the argument lists have not been generalized by that last iteration, a fixed point has been found, and the call cache entry can be closed. Note that the abstract values will not be identical, since non-value-bearing attributes like `atom/code` may change; only the semantic contents of the values and argument lists are compared. This is indicated by the $=_{\text{value}}$ comparison operator.

6.4.4 *Concrete Arguments*

Calls containing only concrete arguments deserve special attention. Especially in the context of numerical applications, optimal compilation requires large amounts of compile-time computations. The FFT algorithm provides a good example: it employs *twiddle factors* (complex roots of 1) that are input-invariant and should therefore be tabulated in advance. A typical "specification-style" algorithm of an FFT algorithm might not describe the twiddle factors as a table of literals, but will simply employ the arithmetic expressions that define them. It is up to the compiler to decide which values can and should be tabulated in advance, and how to organize them.

Especially when computing the trigonometric, exponential and special functions, standard methods heavily depend on iterations to refine their approximate results. The similarity predicate used by the loop detector

```
func fragment1() =
  let
    x1 = fak(30)
    x2 = fak(20)
    x3 = fak(10)
  in
    x1+x2+x3
  end
```

```
func fragment2() =
  let
    x3 = fak(10)
    x2 = fak(20)
    x1 = fak(30)
  in
    x1+x2+x3
  end
```

Figure 6.4: Two program fragments

should not consider these loops as “similar”; otherwise one of the main goals of the compiler (freeing the user from explicit loop unrolling) will not be met.

The $\texttt{similar}_{\text{group}}$ function employed in `ac` is tuned to avoid these problem. It differentiates between concrete and abstract values. When comparing small groups, non-equal concrete values are treated as being different; only in larger groups, comparison is done by base type.

6.4.5 Evaluation-Order Effects

Due to the stack depth limit-triggered generalization process, a new phenomenon occurs: the evaluation order influences the precision of the abstract interpreter.

Consider the approximation behaviour of the function `fragment1` from fig. 6.4 in the context of a loop detector that aborts after a recursion depth of 20 (which just happens to be the generalization threshold).

The computation of `x1` would be approximated, since the call depth of 30 exceeds the loop limit. The result would be *anInt*, and a call cache entry mapping `fak(`*anInt*`)` $\rightarrow$ *anInt* would be established. The subsequent calls to `fak` would therefore hit the cache and return *anInt*, too.

Now consider the re-ordered sequence in `fragment2`:

The computation of `x3` has a depth $<$20 and will therefore terminate with a concrete value. Furthermore, 10 call slots will be filled with the results for `fak(1)`...`fak(10)`. The computation of `fak(20)` will therefore hit the cache at `fak(10)` and terminate. Likewise, the computation of `fak(30)` will hit the cache at `fak(20)`, and deliver a concrete result.

As the example shows, both the precision and run time of the abstract interpreter depend upon details of the evaluation history. This is not a problem that concerns the *correctness* of the abstract interpreter, but its *efficiency* and *precision* – and the efficiency of the final generated code.

6.5 ALDiSP-*specific Problems*

6.5.1 State and Side Effects

A problem that has been ignored up to now is the state. While the matching of calls against call-cache entries is determined by an arbitrary similarity function for the argument lists, *exact* matches are required for the function and context. What happens when a recursive function changes the context (either by performing a side effect, or by changing the exception environment)? This happens rather often, e.g. in stream-processing functions:

```
func streamFromPort(p:aPort) = read(p)::streamFromPort(p)
```

which, by auto-mapping, expands to the equivalent of

```
func streamFromPort(p:aPort) =
  let
    x = read(p)
  in
    suspend x::streamFromPort(p)
    until isAvailable(x)
    within 0.0 ms, 0.0 ms
  end
```

The answer is that this function is not considered recursive at all! That is, it does not call "itself", since it only returns a closure that contains a reference to it. After all, the **suspend** form expands to a **_suspend** primitive function, which installs a closure as a suspension (cf. sections 3.5.3 and 4.4.5). To totally mix up ALDiSP, LF, and the semantics:

```
func streamFromPort(p:aPort) =
  let
    x = read(p)
  in
    apply_primitive(_suspend,
                    [λ (). x::streamFromPort(p)
                     λ (). isAvailable(x),
                     0.0 ms, 0.0 ms])
  end
```

Equivalent things happen to all side-effecting functions, since each side-effect results in a synchronization.

6.5.2 Promises

Still there are the "non-real" side effects, namely promise evaluation and reference lookups, which must be coped with. Accessing previously evaluated references is a side effect that does not modify the state and can therefore be ignored safely. Evaluating promises is somewhat problematic, since a promise should only be evaluated once. The brute-force method of guaranteeing this is to treat a promise like a specialized suspension, i.e. to block the evaluation until the promise is available. The scheduler is then extended to handle promises specially, by preferring them and not advancing the time. The actual implementation employs explicit update and dereference primitives to model promises; this is explained in detail in chapter 8.3.

6.5.3 Recursive Definitions

Fixpoint detection happens at two places in the abstract interpreter. The preceding discussion described the actual application of a recursive function object to a given set of abstract argument values. A different problem is the creation of the recursive function in the first place! The semantics (cf. section 3.4.4) assumes a semantic primitive `fix` that computes the fixpoint of any function. This `fix` is then applied to the semantic function that computes an environment from a set of declarations. The direct iterative approach of implementing `fix` has a severe performance problem: even in the best possible case, the evaluation of the declaration set takes place at least twice. In worst cases, `fix` can cycle forever – e.g., when encountering a definition without a finite fixed point, such as

```
rec
  ones = cons(1,ones)
end
```

Even some simple cases for which a fixed point exist can cause problems. For example, the evaluation of

```
rec
  ones = cons(1,_delay(lambda(). ones))
end
```

does not terminate: each incarnation of the `lambda()` will have a different lexical environment, i.e. the variable `ones` will be bound first to $\perp$, then to the pair (*1*,`Promise`(n)), then to the pair (*1*,`Promise`($n+1$)), and so on. Since unique promises are generated in each iteration, equality can not be guaranteed by the $=_{\text{value}}$ comparison employed in $\texttt{loop}_{\text{approx}}$.

This effect is circumvented if the "tag counter" of promises is reset on each iteration of the `fix` function, since the same unique tags will be allocated for new promises (if the iterations are structurally equivalent, i.e. if promise and suspension tags are allocated in the same order). The need to explicitly reset the counter is a consequence of the side-effecting nature of tags; if the tag counter would be carried along as a part of the context, it would be reset automatically, since the context of the iterated applications stays the same.

There exists an alternative method of implementing recursive environments, which is based on mutable cells.[21] In the semantics of LF reference values can provide this service. A definition like that of `ones` would then be evaluated as if it were of the form

```
n = fresh_tag()          \\ allocate tag
update_tag(n,⊥)          \\ define reference n with some initial value
ones = ref(n)            \\ preliminary value of 'ones'
ones = cons(1,ones)      \\ i.e., cons(1,ref(n))
update_tag(n,ones)       \\ final value of reference n
```

6.5.4 Exceptions

The exception environment is part of the context of evaluation. Most functions do not re-define any exceptions, but at least in theory, it is possible to change the values of exceptions within loops. Currently, no special treatment for such cases exists; i.e. the loop detector detects changes in the exception environments and treats them as unequal values, but it does not attempt them.[22]

6.5.5 Function Values

Higher-order languages pose a special problem for an abstract interpreter: they allow treating functions as values. Function values are notoriously hard to handle, since even the most basic operations are not defined on them: for example, it is – in general – impossible to compare functions for equality.

The most immediate problem is exemplified by the following example:

```
func f(x)(y) = x+y
func g(x)(y) = x-y
func h(x) = if x>42 then f(x+27)
                     else g(x-45)
            endif
```

The `if` expression delivers a value that denotes one of two functions. When called with a non-concrete argument for `x`, the abstract interpreter has to find the least upper bound of two functions. In the ALDiSP type hierarchy, there is a type "aFunction" that denotes all functions, but an abstract value that would denote only "any function" cannot be considered precise enough, since any application of it would have to return a $\top$ result, including a "top state" – something which the current abstract interpreter is unable to handle.[23]

[21] It is not necessary that the cells be arbitrarily mutable; it is enough that they can be updated once. The cells have an initial value of $\perp$, which is then replaced by a more specific value. I know of no language that supports such an update concept.

[22] This is another possibility to drive ac into non-termination.

[23] The state is not a data object, but a heterogenous collection of data objects maintaind in semantics-level data structures. To allow abstract states (and, much more useful, partially abstract states), a type "State$\subset$`Obj`" would have to be introduced. While possible in theory, this was considered too much trouble.

There are a few approaches to merging function-valued objects:

- Arbitrary sets: as explained in the beginning, the most general abstract value concept is that of sets of arbitrary values. As long as these sets stay small, using them is possible. Nothing forbids sets of function-typed values.

- On-the-fly code generation: when the functions that are to be merged take the same number of arguments, a new function can be generated. In our example, this would correspond to a transformation of the function `h` into:
  ```
  func h(x) = (λ(tmp).
                (if x>42 then f(x+27)
                         else g(x-45) endif)(tmp))
  ```
  If the functions do *not* take the same number of arguments, as in
  ```
  func a(x) = ...
  func b(x,y) = ...
  ... if ... then a else b endif ...
  ```
  they can be overloaded:
  ```
  func a(x) = ...
  func b(x,y) = ...
  ... _overload(a,b) ...
  ```
  A conditional is not necessary to distinguish between the different cases; the different arities are sufficient to disambiguate any application of the overloaded result function.

- Closure generalization: if the functions that are to be merged are different closures derived from the same lambda expression, only their environments differ. Since environments are bindings of variables to abstract values, they can be merged. A new closure tag has to be allocated for this merged function, so that the components can be specialized separately. A problem of this approach is that code has to be generated that implements the operation
  ```
  closure_new = if cond then closure_true
                        else closure_false
                end
  ```
  This cannot be directly represented in the back-end representation, since closures are not represented as first-class values. They are instead modelled as environment tuples held in a global table, and indexed by the closure tag. To model the conditional closure assignment, a new primitive must be introduced, for example
  ```
  _conditional_copy_closure(tag_new,tag_t,tag_f,cond)
  ```

- Preprocessing: purely syntactical local transformations can remove some occurrences of this problem in advance. This amounts to recognizing some typical special cases.

As there exist only a finite number of lambda expressions in each program, the "closure generalization" approach can be taken to its extreme by writing an "over-function" that encompasses all other functions. Its first argument would be a selector that indicates the function to be computed. For an example, consider the three functions

```
func f(a) = g(a+10)
func g(b) = h(b+20)
func h(c) = c*21
```

They can be merged into one function that takes an extra selection parameter:

```
func fgh(x)(abc) =
   if x=="f" then fgh("g")(abc+10)
   [] x=="g" then fgh("h")(abc+20)
   [] x=="h" then abc*21
   ⋮
```

There are some problems due to the handling of different arities, but these can be avoided by using tuple values instead of argument lists, or by filling up all argument lists with dummy arguments.

The implementation of `ac` implements function merging by on-the-fly code generation. Attempts to merge functions of different arity are flagged as errors.

6.5.6 Generalizing Numeric Types

In ALDiSP, numeric types are parameterized with their width. A consequence of this is that the "base type" of a numeric literal is not unique. For example, "17" can be an `nBitInteger` of any width ≥ 6, or an `nBitCardinal` of any width ≥ 5. For the similarity measure that is built into the loop detector, this raises the question whether type matches have to be exact or not. For example, are `nBitInteger(5)` and `nBitInteger(6)` to be considered "nearly the same"? If this is the case, is `nBitInteger(5)` "nearly the same" as `nBitInteger(1000)`? The current implementation demands exact matches on numeric types, but has special provisions for literals (cf. section 6.4.4).

6.5.7 Raising and Catching Exceptions

Exception values in ALDiSP are bound dynamically. When an exception is raised and **returns**, it "travels up the call stack" and is caught at the first matching **guard** declaration. Since there is no explicit call stack in the semantics, the diverse semantic functions have to cooperate to achieve this effect. At the LF level, the ALDiSP **guard** construct has been separated into an exception-environment declaration and an `Lcatch` expression.

While it is easy to implement the exception binding behaviour – it suffices to introduce an extra environment component into the context, together with the appropriate primitive functions – the raising and catching of exceptions creates some problems.

In the standard semantics, raised exceptions are modelled by `exception`(*tag*, *obj*, *state*) tuples. All semantic rules cooperate by checking all sub-results for exception-ness. These checks are implemented in the $\texttt{deref}_{\texttt{obj}}$ function. Only the `Lcatch` form can transform exception tuples into "real" results.

In the abstract interpreter, these exception tuples cannot be treated like ordinary values. Extending the abstract value domain with special forms `exceptionOrValue`(*tag*, *obj*, *state*, *realValue*) would indirectl re-introduce "general set" abstract domains with all the related efficiency and termination problems, since multiple exceptions require exception value sets. An expression such as

```
switch([Int(2)] c )
 of case 0: 42.0
    case 1: raise Overflow(17)
    case 2: raise Round(19.4,1)
    case 3: raise StupidExample("Gotcha!")
end
```

would have to be represented by listing all three possible suspension tags, plus their differently typed and sized tuples.

It hardly makes sense to complicate the very basics of the abstract interpreter just to guarantee a correct handling of exceptions.[24] Instead, a "hack" is used.

[24] Exceptions will hardly occur in the inner loops of realistic real-time code anyway; they are mostly needed to deal with one-time initialization and I/O problems.

First, whenever an exception value is merged with a "real" result value, it is treated like $\perp$ and completely ignored.

A "catch" expression is treated almost like an identity function by the abstract interpreter, but the return value is *generalized* to its base type. This follows from the premise that an exception handler will return a value of the same type as that which would be returned if no exception had occured.

The abstract interpreter thus behaves as if dynamic occurence of an `Lcatch` might actually catch an exception. The result is generalized, since there is no information available about the caught object. It can be assumed that the caught object has the same basic type as the "regular result", so a simple generalization up to the base type suffices. The generalization is needed for correctness, as can be seen from the following example:

```
guard if x<>0 then stop(x) else 42 end
   in stop(x) = return x
end
```

If the `guard` were treated as an identity function, the result would be `42` regardless of the value of `x`. Because of the generalization, an `Integer(0)` is returned.[25]

A more sophisticated strategy can be implented by keeping track of a global "exception state" indexed by the exception tags. Upon encountering an `Lcatch`, the respective slots (of the exceptions that can be caught by it) would be cleared; the `_mk_exc` primitive would store the state and object of the exception in its slot, so that a type-safe merge can be done upon return to the `Lcatch`. In the case where no exception has occured, no generalization is necessary. If more than one exception for a given slot occurs, the respective objects have to be merged.

[25] There is a small problem in this example: the type of `x` should be used to determine the type of `42`, since the latter is a literal and therefore of "malleable" type.

Chapter 7

The Abstract Scheduler

The "imperative" aspects of ALDiSP programs are defined in terms of a state model in which a global scheduler controls which suspensions to evaluate and when. The scheduler also controls all I/O operations. It determines the order of suspension evaluation as well as the relation of this order to the virtual time. The scheduler is part of the compiler; it has no access to the actual execution times.

The scheduler exerts its control of time by issuing *time-advance* state transitions. In the code that is later emitted, each time-advance is translated into a *wait instruction* that is intended to synchronize the instruction stream with a real time clock. All I/O operations are performed at points of time-advance – i.e. all I/O events that have accumulated between two time-advances are performed simultaneously at the latter one. Each time-advance is therefore a synchronization point.

The abstract scheduler differs from the standard semantic scheduler in in that it also introduces *renamings* between similar states at different points of time, and thus collapses the tree of all possible state transitions into a finite graph. The set of state transitions that form the edges in this graph is later used to provide the "skeleton" of the emitted code (cf. chapter 8).

7.1 Suspensions and other Schedulable Entities

The standard semantics of the scheduler is given in section 3.7.2, as part of the definition of $\texttt{eval}_{\text{program}}$. Here, its implementation shall be outlined.

In the ALDiSP language model, suspensions themselves do not occur as semantic entities of computation. Instead, *references* to suspensions are employed. References (or *reference objects*, for they form a subset of the domain of values *Obj*) have multiple uses: they may also refer to blocks, promises, and events. The state consists of a set of definitions for these references. In the current implementation, there are the following types of reference definitions (semantic domain *RefDef*):

- An `Ususp` is an unevaluated suspension. The evaluation of a `suspend` form (in LF the `_suspend` primitive function) generates a reference that has a `Ususp` definition.
- A `Wsusp` is a waiting suspension. It contains a "time frame", i.e. information about the interval of time during which it has to be executed. When the scheduler determines that the condition part of an `Ususp` has become true, it is transformed into a `Wsusp`. Each following time-advance decrements the "time frame counters" in all `Wsusp`s.
- An `EvRef` is an evaluated reference, i.e. a value. After a `Wsusp` (or an `Uprom`) has been evaluated, an `EvRef` holds the result.

- A `Block` is the result of a blocking access to a reference. Blocks can be treated as special cases of `Ususps` that have a condition of the form `isAvailable(x)` and a time frame of `[0,0]`. Blocks differ from all other reference definitions in that they contain semantics-level continuations instead of parameterless closure objects.
- A `Uprom` is an unevaluated promise. The first access to it forces its evaluation, generating an `EvRef`.

Besides these "computational" reference values, there are pseudo-suspensions used to model I/O. These have the form `Event(`*device*, *kind*, *obj*`)`. The *device* names the register or port that is accessed by the event. The *obj* is the object to be written, or (in case of a read or test operation) a dummy. There are three *kinds* of event:

- An `Input` event is generated from primitive functions like `readPort`. If the input device is a register, the input is performed at the next time-advance; inputs from a port device are performed at arbitrary times.[1] After the event has been performed, the `Input` is replaced by an (`EvRef` pointing to an) abstract input value. Until then, all accesses to the (reference pointing to the) `Input` will block.
- An `Output` event likewise models an output, i.e. a true side effect. Output to registers is performed at the next time-advance; output to ports may wait for an arbitrary amount of time. The return value of an `Output` event is a dummy; it can be used (using the `isAvailable` primitive) to detect whether the output transaction has been performed or not.
- A `Test` event tests the accessibility of a port; it evaluates to either `true` or `false` without blocking. While such a test could be modelled by a non-blocking primitive (since the accessibility of an I/O port is a property of a given state), it is better modelled as an event so that the decision is lifted to the scheduler level.[2] In the abstract scheduler, such a decision will "fork" the set of successors to a state into those in which the test return `true` and those in which it returns `false`.

The core of both the standard and the abstract scheduler is the `succs` function that, given a state A, computes the set of all its successor states.

Depending upon the contents of a state, none, one, or a large number of the following state transitions may be used to derive a valid successor state:

- An `Ususp` with condition evaluating to `true` can be *advanced* to a `Wsusp`.
- An `Ususp` with condition evaluating to `aBool` can be *assumed* to return either `true` or `false`.
- A `Wsusp` with a timing interval that includes the current point of virtual time can be *evaluated*. If the upper limit of its timing interval is 0, it *must* be evaluated before the next time-advance, since "now" is that last chance to evaluate it.
- Time can be *advanced*; all pending `Input`, `Output`, and `Test` events are then *performed*. This may generate more than one result state, if there are asynchronous events present, for which the actual execution time can not be determined at compile time.

The only real restrictions imposed on the scheduler are that `Ususp`s should be advanced to `Wsusp` status as early as possible (otherwise, the specified timing interval starts too late), and that no time-advance is possible when there is a `Wsusp` that must be scheduled at the current point of time.

[1] "Arbitrary" means that the programmer has no guarantee about when the event is performed. I/O that employs asynchronous registers is transformed into non-deterministic state transitions; when a state contains k asynchronous events at the moment of its time-advance, there will be 2^k successor states with all different combinations of performed/still pending events.

[2] Otherwise, each abstract time-advance would have to create 2^n successor states, where n is the number of I/O ports that are referenced anywhere in the program. With `Test` events, these states are constructed on-demand.

7.2 Non-Determinism

There is a lot of non-determinism in the abstract scheduler, especially where the evaluation of `Wsusp`s is concerned. When a `Wsusp` has a non-point range, i.e. a range of the form $[m \ldots n], m < n$, it may be evaluated at $\frac{n-m}{\delta_t}$ points of virtual time (where the time is assumed to be quantized by some duration δ_t). Furthermore, even if all ranges in a program are point-like, there may be situations in which more than one `Wsusp` can be evaluated at one point in time, so that the scheduler has to choose an order.

It would be nice if a program that has a well-defined I/O behaviour, i.e. in which no to output operations have overlapping performance intervals, would act the same regardless of the ordering that the scheduler imposes on it. Non-deterministic programs that do not employ I/O are however simple to write. Consider the following expression, which may return either `1` or `2`:

```
let
  s1 = suspend 1 until true within 0.0 ms, 0.0 ms end
  s2 = suspend 2 until true within 0.0 ms, 0.0 ms end
in
  suspend if isAvailable(s1) then s1 else s2 end
  until isAvailable(s1) or isAvailable(s2)
  within 0.0 ms, 0.0 ms
  end
end
```

It would be possible to let the scheduler enumerate all possible states. A later phase could then, guided by a cost function, choose a particular execution path, and warn the user about non-deterministic behaviour.

The problem with such an exhaustive approach to non-determinism is the size of the state space: it would grow rapidly. If the program is I/O deterministic, the growth is less than exponential, since there are "synchronization points" at which similar states will be merged together again.

An example is shown in fig. 7.1, which graphically represents the state graph of the following program:[3]

```
func demo() =
 let
  s1 =  suspend 1 until true within 5.0 ms, 10.0 ms end
  s2 =  suspend 2 until true within 5.0 ms, 10.0 ms end
  s3 =  suspend 3 until true within 5.0 ms, 10.0 ms end
 in
   suspend
      demo()
   until isAvailable(s1)
      or isAvailable(s2)
      or isAvailable(s3)
   within 5.0 ms, 10.0 ms end
 end
```

The scheduler can choose between any of the suspensions `s1;s2;s3` at first and any of the remaining three afterwards. Scheduling sequences like `s1;s2` and `s2;s1` lead to the same result – `s1` and `s2` evaluated, the others not evaluated – and can be shared. This sharing is expressed via a renaming ("loop") transition.

7.3 Sequentializing the Schedule

A lot of the non-determinism can be eliminated at compile-time by choosing an (arbitrary) scheduling order. This early ordering minimizes the "fan-out" of states, and thus cuts down the size of the state graph. The

[3] The state space diagram was generated directly by `ac` via the `-graph` option. The output is a graph description that was layed out afterwards by the `GraphEd` tool with minimal manual intervention.

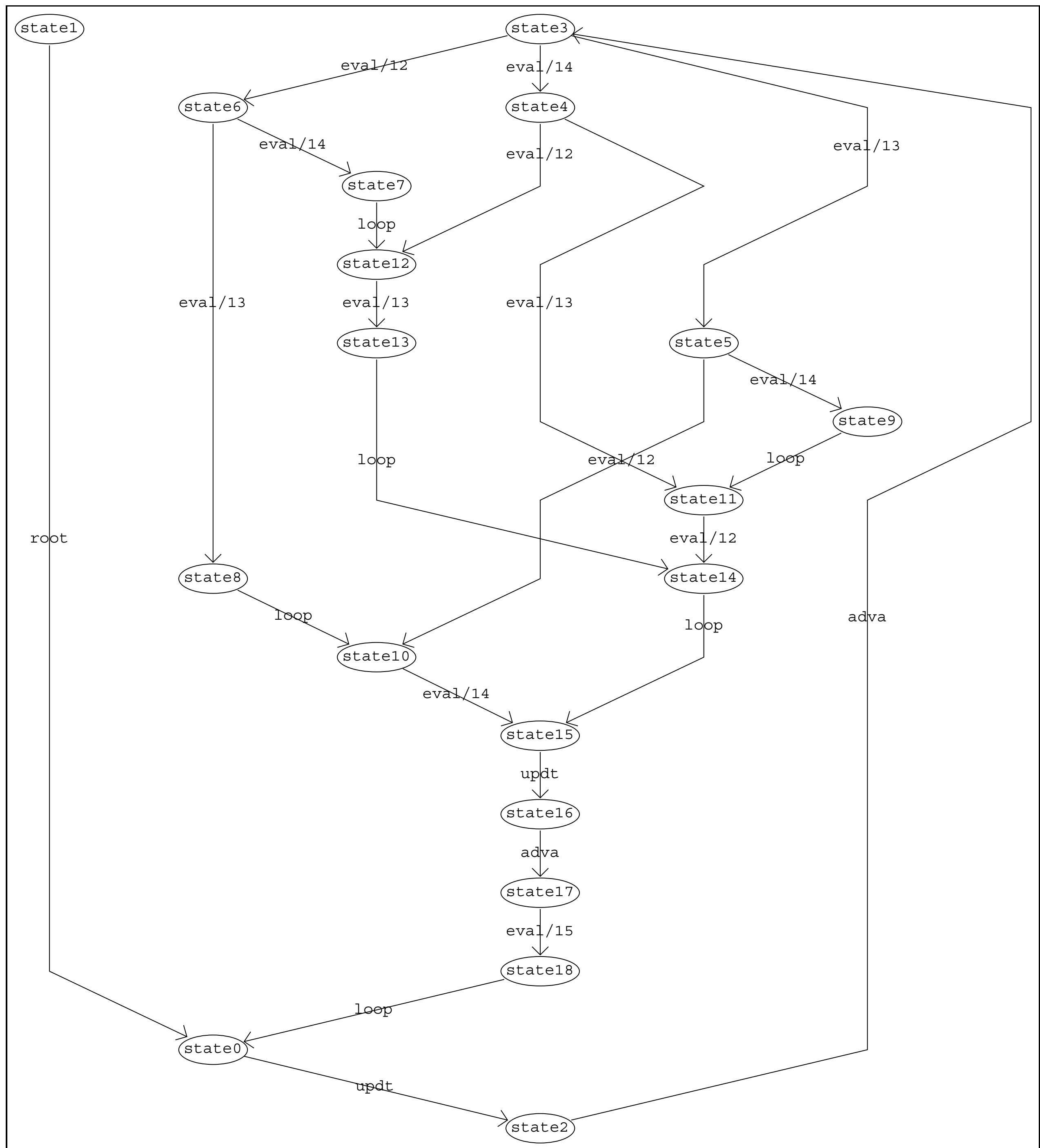

Figure 7.1: Non-Determinism and Merging in a State Graph

state graph takes on a more linear form, and contains only those forks that are a semantical necessity.

In the current implementation. the transitions are sequentialized by the following rules:

- If there are evaluable `Block`s, they are evaluated. Otherwise,
- If there are advanceable `Ususp`s, one is choosen and advanced. Otherwise,
- If there are evaluable `Wsusp`s, one is chosen to be evaluated.[4] Otherwise,
- If there are assumable `Ususp`s, one is assumed (with two result states). Otherwise,
- If there are no `Ususp`s or `Wsusp`s at all, there are no successors (and this part of the state tree terminates). Otherwise,
- Time is advanced.

The only non-determinism left is in the "assumptions" of `Ususps` with an unknown condition, and the I/O assumptions that are introduced when asynchronous I/O is performed during the time-advance.

7.4 Garbage Collection

States can be *garbage-collected* at compile time. Garbage collection (GC) disposes of objects that are guaranteed not to be used by any future computation. GC does this by computing a conservative approximation to this set of "unneeded objects": it enumerates all objects that are structurally accessible ("alive") from a known set of "roots" which are part of the current program state, and eliminates all objects that are inaccessible ("dead"). The objects of interest when garbage collecting an ALDiSP state are the reference definitions.

The roots, i.e. the references that are assumed to be "alive", are

- all `Input/Output/Test` events
- all `Ususp`s[5]
- all `Wsusp`s

The only references that may be removed by the garbage collection because they can become inaccessible are thus `EvRef`s and `Uprom`s.

Garbage collection is a necessary prerequisite for successfully finding loops, since it removes "old" `EvRef`s that would, without GC, accumulate. Without this trimming, states would grow over time, and never be considered similar by the state loop detector, since the no structural correspondence would be found for the "dead" `EvRef`s.

States that contain `Block`s cannot be garbage collected, since a block encloses an opaque interpreter state, which may address arbitrary references. This is another reason to minimize the number of `Block`s generated by the abstract interpreter.

[4] This greedy scheduling of `Wsusp`s can be replaced by a randomized placement strategy with a command-line option.

[5] Only `Ususp`s that have a condition that can possibly evaluate to `true` at some time are really roots. Since this condition cannot be tested, it is ignored. One way to bring `ac` to its knees is therefore to produce suspensions with a `false` condition; these will never be advanced to `Wsusp` status, nor removed by the garbage collector. It might be possible to determine whether a condition depends upon the current state, but such an analysis is probably not worth the complexity.

The garbage collector is implemented using a straight-forward mark-and-sweep algorithm: beginning with the roots, all reachable objects are enumerated. All references that were not reached are removed. Since the garbage collection happens at compile time only, no special tricks are needed to speed it up.[6] To find the references reachable from a given root, a function has been written that enumerates all sub-values of a value. While this is easily done in the non-abstract value domain, abstract values pose some slight technical challenges: Abstract values can contain attributes that are themselves values, or contain references to values. For example, the `atom` and `code` attribute introduced in chapter 8 may contain literal values, which can be reference objects. A code walker is needed to find these references.[7] Also, the compiler phase that generates these code fragments (which is mostly the reconstruction phase, cf. chapter 8 ought to minimize the amount of objects that are "touched" in a piece of code.

Garbage collection is a time-consuming process. To minimize these compile-time costs, collections are only performed after each time-advance.

7.5 *Finding Loops*

Beginning with the initial state, a tree of states will be enumerated by successive application of the `succs` function. For most DSP applications, this tree will be of unbounded size, representing a nonterminating program behaviour.

To create a finite schedule, a *loop detection* scheme is employed that finds *similar* states. There is a resemblance between the state loop detector and the function loop detector that approximates fixpoints in the abstract interpreter, but it is only superficial. The current state loop detector does not involve any approximation, hence needs no iteration steps.[8]

A state A is "similar" to a state B if there is a *valid renaming* that maps A to B. A *renaming* between two states establishes a structural correspondence between *all* the data structures that make up the states. To represent such a mapping, data objects must have a unique identity. Only closures and references have such identities; the state mapping is therefore represented as a mapping from tags to tags. A renaming is *valid* if each value is mapped onto a value that is *structurally equal*. Scalar values are considered *structurally equal* if they have the same base type; non-scalar values (i.e. data structures) must also have the same "shape" and structurally equal components. Determining structural equality thus becomes non-trivial when recursive structures (such as function environments) are considered. Functions are considered structurally equal if they are derived from the same function definition (this is easily determined by comparing their `Lambda` expression tag), and their environments contain structurally similar values.

The abstract scheduler enumerates the state tree in a lazy manner. It maintains a list of "front" states, which initially contains only the start state. Each scheduling step consists of computing the set `succ`(*front*) of all possible successor states of all front states. These successors are the new front of the state space. The new front is simplified by searching for similar states within the front and amongst the previous states. When a number of states in the front are found to be similar, one is chosen as representative, and the others are mapped to it.[9] If a similar state is found that is not in the front, a mapping to the earliest state is introduced.[10] This corresponds to finding a loop in the program.

[6] A large body of literature describes tricks and techniques to increase the speed and decrease the memory consumption of runtime systems and runtime data encodings that support garbage collection or one of its alternatives, like reference counting or linear logic.

[7] In general, a traversal scheme for each attribute is needed; in the actual `ac`, only `code/atom` attributes may contain values.

[8] A later version might incorporate a state generalization scheme as a means to further reduce the state space.

[9] This amounts to partitioning the front set into a set of equivalence sets.

[10] There may be more than one such state in the history of the graph, in which case the "in-between" states will already be connected to the earliest similar state. Looping back to the earliest state (instead of an arbitrary earlier state) removes "jump chains".

Each time a mapping is found, one entry is deleted from the front. When the front finally becomes empty, the abstract scheduler terminates. There is no guarantee for this to happen, so maximum sizes can be introduced for the state space and the current front to force a termination.[11] Such a size limitation has been proven to be a necessary debugging aid.

After the scheduler has terminated, the state graph is given to the reconstruction phase (cf. chapter 8), which recreates the program in the form of executable code.

As with garbage collection, loop detection does not work on states that contain `Blocks`. A state space in which all states contain one or more `Block`s can therefore not be transformed into a finite graph.

7.6 Alternative States

To simplify the abstract interpreter, the design decision was made that each run of the interpreter must produce exactly one new state. This is important if one considers how to treat an expression like

```
let runTimeVal = read(inPort)
in
 if runTimeVal>42 then suspend ... until ... within ... end
                  else suspend ... until ... within ... end
 end
end
```

Depending upon the run-time value of "`runTimeVal>42`", which is an `aBool` in the abstract interpreter, one suspension or the other is to be evaluated. Quite differently shaped states may emerge from such a situation. There is no obvious way to create an "abstract state value" that describes these alternative. The place to handle this decision is the scheduler, which is already equipped to handle assumptions.

Therefore, a new reference type `IfSusp` is introduced that models this behaviour: an

`IfSusp(cond,`$\texttt{expr}_{\text{true}}$`,`$\texttt{expr}_{\text{false}}$`)`

will be returned to the scheduler level. The two *expr*s are ad-hoc closures that encode the alternatives of the conditional. The modifications to the scheduler are minimal; only the `succs` function has to be extended with a rule similar to that needed for assumable `Ususps`.

7.7 Blocking

One all-pervading behaviour of ALDiSP is the automatic dereferencing of references, and the resulting "blocking" behaviour that occurs every time an unavailable reference is accessed. For example,

```
3+(4+(5+suspend 6 until true within 0.0 ms, 0.0 ms end))
```

will create 4 suspensions that are linearly dependent upon the innermost expression. The resulting state space, and the generated code, are equivalent to that which would be generated for the explicit variation:

[11] On a Sparc-2 with 32M memory, a run that enumerates 50 states can easily fill up all the core memory. It would be useless to proceed after this happens, as thrashing reduces the performance to an unacceptable level.

```
let
  tmp1 = suspend 6 until true within 0.0 ms, 0.0 ms end
in
  let
    tmp2 = suspend 5 + _deref(tmp1)
     until isAvailable(tmp1) within 0.0 ms, 0.0 ms end
  in
    let
      tmp3 = suspend 4 + _deref(tmp2)
       until isAvailable(tmp2) within 0.0 ms, 0.0 ms end
    in
      suspend 3 + _deref(tmp3)
        until isAvailable(tmp3) within 0.0 ms, 0.0 ms end
    end
  end
end
```

While this behaviour is fine for specification purposes, it uses a lot of resources: At the scheduler level, there will be four states involved in the evaluation of this expression: the evaluation of each suspension necessitates one state transfer. Each of these states will be compared with previous states to find loops, and will be the target of comparisons with all future states.

It is clearly necessary to minimize the impact on blocking behaviour on the abstract interpreter by reducing such chains of blocks. For example, transformation of the example expression to

```
let
  tmp1 = suspend 6 until true within 0.0 ms, 0.0 ms end
in
  suspend 3+(4+(5+_deref(tmp1)))
    until isAvailable(tmp1) within 0.0 ms, 0.0 ms end
end
```

would approximately halve the evaluation costs. The current interpreter contains two ad-hoc optimizations that will catch some situations like these. The first optimization is located at the top of the $\texttt{eval}_{\mathrm{expr}}$ function: Whenever an $\texttt{eval}_{\mathrm{expr}}(e, C)$ evaluates to a block that waits for a suspension already present in C, the result is discarded, and an explicit suspension is created instead. The second optimization does essentially the same for sequences.

At the interpreter level, the problem is one of software complexity and modularity: the need for "forced" values, i.e. values that are guaranteed not to be references, occurs frequently:

- When applying strict primitive functions (some functions, like `isSuspended` or `cons`, are nonstrict)
- When applying a function – the function itself has to be known to be applied,
- when testing the condition of an `if` expression
- when testing for a type, especially when resolving an overloaded function application

So as to centralize the handling of blocking behaviour in the interpreter, it would be nice to have a function `force` that is guaranteed to returns a "real" value. `Force` will have to introduce a call to `_deref` if the value is a reference to an `EvRef` (this is the simple part), and it will evaluate `Ususp`s and replace them by `EvRef`s. But what is to happen if the reference points to a real suspension[12]? Luckily, the interpreter is implemented in SML, which support exceptions. `force` will raise an exception and abort, and the exception

[12] That also includes I/O events.

will be handled "as late as possible", i.e. at the earliest point of time possible. What is this earliest point of time? Obviously, it should be a point where it is reasonably easy to insert a suspension; best a point where it is done anyway. One of these points is the `if` handler, which will have to raise a `IfSusp` if the states that result from the arms of the `if` differ. Another place has to be the point where the suspension (that causes the blocking) is defined, i.e. the piece of code that models function application. Indeed, the blocking exception may not backtrack beyond any point where the state has been changed, since then the state that surrounds the block would be different from the state in which the block is handled, which could make it useless.

The correct handling of blocks can be developed by looking at all possible cases:

- `Lsimpledecl(`*pos*, *symbols*, *expr*`)`: Here, the block can occur when evaluating the expression. The fact that the block occurs here implies that the evaluation of the expression did not introduce a side effect. Therefore, the block exception can be passed on; i.e. nothing has to be done at all.
- `Llocal(`*pos*, *decls*`)`: this is translated into `Let/Lsimpledecl` combinations.
- `Lambda(`*pos*, *tag*, *params*, *body*`)`: in a naive interpreter, no suspensions can occur here. De facto, they can, since the type expressions in the parameter declaration list are evaluated at this time, so as to prevent unnecessary multiple evaluation. Type expressions are restricted to be side-effect free, so it is an error if a Block occurs here.
- `Lvar(`*pos*, *sym*`)`: Variable access cannot block.
- `Lit(`*value*`)`: Literal access cannot block.
- `Lapp(`*pos*, $[expr_1, \ldots, expr_n]$`)`: if one of the expressions blocks, the application is translated into the canonical form

```
let tmp1 = expr1
        ⋮
    tmpn = exprn
 in
    tmp1(tmp,...,tmpn)
end
```

- `Lcc(`*pos*, $expr_{type}$, $expr_{value}$`)`: if the $expr_{type}$ evaluates to "Obj", nothing happens at all (this is a consequence of the semantics and guarantees that null-`Lcc`s do not change the evaluation order. Again, the `expr_type` itself is not allowed to block or to change the state. If the `expr_value` blocks, the block can be passed along, since the `Lcc` would not change the state.
- `Lcond(`*pos*, $expr_1$, $expr_2$, $expr_3$`)`: if $expr_1$ blocks, the block can safely passed up. If the result is of $expr_1$ is `true` or `false`, $expr_2$ or $expr_3$ can be evaluated in isolation, the `if` can be ignored. If $expr_1$ is `aBool`, both alternatives have to be evaluated and merged. If the alternatives return with differing states, an `IfSusp` will be returned (that was already mentioned). If one alternative blocks, and the other not, this is considered to indicate "different states", too. If both alternatives block, the `if` is translated into be the canonical form

```
let
    tmp = expr1
 in
    if tmp then expr2
            else expr3
    end
end
```

and the whole `if` is considered to block.

- `Lselect`(*pos*, *expr*, *alternatives*, *default*): this is considered equivalent to a set of nested `if`s.
- `Lcatch`(*pos*, *strings*, *expr*): this can simply pass the block on.
- `Let`(*pos*, *expr*, [$decl_1, \ldots, decl_n$]): There may be two situations: one of the `decl_i` may block, or the `expr` may block. A blocking `decl_i` may not block the whole `Let`, since it in "naive" interpretation, the resulting suspension would be bound to a variable, not blocking the further evaluation at all. Ergo, such a suspension has to be created and assigned to the respective variable, before proceeding.

  The other possibility is that of a blocking `expr`. If the evaluation of the *decls* did *not* change the state, the block can be passed on. Otherwise, if one $decl_i$ *did* change the state (here, state changes due to the introduction of blocks do not count!), the `Let` can be splitted as

```
let decl1
      ⋮
     decli-1
     decli
in
  let
    decli+1
      ⋮
    decln
  in
    expr
  end
end
```

  and a suspension can be wrapped around the inner `Let`. This way, a minimum of work is done beyond the last state change, and blocks that might have occured in $decl_{i+1}$ ... $decl_n$ may be deferred.

  Lastly, if the `expr` blocks and all state-changes done in the *decls* are due to blocks, the block can be passed up.

- `Lseq`(*pos*, $expr_1, \ldots, expr_n$): Sequences are used especially for the modelling of blocking behaviour: if an $expr_i$ returns a suspension, the rest sequence has to wait for its evaluation. Since the very introduction of a suspension will have changed the state, this has to be implemented faithfully, as equivalent to

```
let
  tmp = expr1
in
  suspend seq expr2 ... exprnd
    until isAvailable tmp
  end
end
```

  If a block occurs during the computation of an expr (usually the first expression, since all but the last one will return suspensions anyway, and thus be removed from the head of the expression list), a suspension has to be introduced to model it.

Chapter 8

Execution Trace Reconstruction

This chapter describes how the program is reconstructed once abstract scheduling and interpretation have finished.

The abstract scheduler has generated a state graph; the abstract interpreter has written a call cache. The state graph consists of state descriptions and annotated transitions between states; the call cache contains an entry for every function call made during the AI. Each state of the state graph is characterized by a set of reference definitions. Each call cache entry contains the arguments to the function and the residual code that was generated by its execution.

Starting with state graph and call cache, the reconstruction phase re-creates the program. The reconstruced program is represented in *Code Form*. The Code Form (CF) started out as a subset of the Lambda Form, but evolved into a separate intermediate representation. The Code Form is designed to make data-flow properties of the program explicit, and to introduce names for all temporaries.

The "shape" of the reconstructed program is determined by the state graph. Each state and state-transition is modelled by a function; a state transition function s_x->s_y that leads from state s_x to state s_y is called within the function that represents s_x and ends with a tail-recursive call to the function that implements s_y.[1] The program as whole is the set of state- (and state-transfer) function definitions, amongst which is one distinct initial state.[2]

There are different kinds of state transitions (cf. chapter 7.1), and a different kind of code is generated for each of them. Transitions can be purely administrative (`Ususp` to `Wsusp` advancement, no code involved), branches (`Ususp` assumptions and `IfSusp`s, a conditional is generated), evaluations of lambda closures (for which a function application is generated), or continuations of `Blocks`.

The application of a function closure (caused by a `Wsusp` evaluation) may introduce many reference definition updates as a "side effect". Reference definitions are initially modelled by modifications of one *state variable* that is passed around as argument to the state functions; eventually, this one variable is split up into a separate variable for each reference definition.

Most of the reconstruction effort takes place during the abstract interpretation phase of the compiler. The abstract *Result* type used by the AI is extended with a `code` attribute. This attribute contains a Code Form fragment which, when executed at run-time, will compute the concrete value which the abstract result approximates.

[1] Later optimizations merge state- and state-transition functions by including the body of each state-transition function s_x->s_y in the code of the state function s_x that calls it. Also, linear sequences of state functions are collapsed into one function. Conceptually, it is simpler to consider the state- and state-transition functions separated.

[2] There is also one distinct exit state generated for programs that have terminating state paths.

Since design considerations of the reconstruction stage determine the restrictions and conventions of the Code Form, and thus those parts of the semantic functions that generate the `code` attribute, these have not been mentioned during previous chapters. Code generation is orthogonal to the (standard or abstract) semantics, since it does not change the execution behaviour.

This chapter starts off by explaining the Code Form, and the conventions regarding closure and state representation. A formal semantics is given. An example follows that exercises the code generator on a variety of tasks. The resulting code will contain many inefficiencies. A set of inter-function transformations (*inlining*, *grouping* and *tabulating*) and basic block optimizations (*copy propagation*, *reference-* and *tuple-tracing*) are explained next. All transformations are applied to the code that was generated in the example.

A second example tackles the specific problems of code generation for `Block`s, and explains some heuristics that minimize the costs of resuming a blocked computation.

The two examples cover all major aspects of code generation as performed during the abstract interpretation. A description of the state-functions follows; these are not generated during AI, but from the state graph that is generated by the abstract scheduler.

A concluding section contains a semi-formal description of the code generation tasks that are performed by the semantic functions in the abtract interpreter.

8.1 The Code Form

The Code Form (CF) has some properties similar to the “A-Normal Form” described in [40]. Both forms have many characteristics of continuation-passing style (CPS), but avoid the use of explicit continuations whenever possible. In the first few incarnations of the back-end of the compiler, a subset of the Lambda Form was employed in the reconstruction and code generation phase. This made it necessary to distribute many semantic validity checks over the whole code generation phase. It was also cumbersome to express some concepts (notably multiple-output expressions and the explicit state variables) in the context of LF. The introduction of a new intermediate representation made it possible to encode the semantic restrictions of the post-AI program directly into the structure of its data type.

8.1.1 Differences between CF and LF

When compared to the Lambda Form, the main properties of Code Form expressions are:

- The CF has no expressions to denote type-casting (`Lcast`) or type-checking (`Lcheck`).
- The CF has no “sequential statement” expressions (`Lseq`).
- All arguments to function applications, and the conditions of conditionals, are *atomic*. As a result,
- all intermediate results are bound to variables, and
- all variables are of a known type.
- The intermediate results are bound by declaration sequences that correspond to `Let(Lseq(...)...)` combinations in LF.[3]

[3] This is the most important difference to CPS, where intermediates are passed on as arguments of continuation functions.

- State is made explicit by passing "state variables" to those function that access reference values, and by returning a new state from those functions that modify it.[4] The state variables are single-threaded: there exists only one living state variable at any point of time. Once a new state variable has been introduced, the preceding one is never again referenced. The primitive that implements reference lookup takes the current state variable as argument; the primitive that implements reference update takes the current state variable as argument, and generate a new state that is immediately bound to a fresh variable.
- All declarations are lexical.
- Those dynamic (exception) declarations that cannot be modelled by function specialization are passed around as part of the state.
- The Code Form does not contain any equivalent for `Lambda` expressions. All function definitions are installed at the top level, i.e. their free variables must be part of the top level. The evaluation of a `Lambda` expression is modelled by the creation of a tuple that holds the lexical variables needed when the function will be applied. To apply a function, both this tuple and the name of the global function are needed.

The definition of "atomicity" is based on the number of reduction steps needed to evaluate an expression.[5] If, for a given expression, the number of reduction steps is guaranteed to be less than k, that expression is called k-reducible. Variables and literals are 1-reducible. Primitive function calls with n 1-reducible arguments are considered $n+2$-reducible. Applications are estimated to be $n{+}m{+}4$ reducible, where n is the reducibility of the arguments, and m the reducibility of the called function.[6] The reducibility of a conditional statement is the maximum reducibility of each of its paths. All expressions that contain no function applications have a finite reducibility. If an expression contains an application, reducibility cannot (in general) be statically determined, since the evaluation may not terminate. A nonterminating expression would have evaluation time ∞.

In `ac`, terms are deemed atomic if they are 1-reducible.[7]

8.1.2 Abstract Syntax of CF

The abstract syntax of the Code Form is as follows. The `code` attribute of each abstract result contains a `Code` expression:[8]

```
datatype Code = Code of {decls: Decl list,
                         exprs: Atom list,
                         state: StateBehaviour}
```

A `Code` expression consists of a list of declarations, a list of atomic return values, and a description of the "state behaviour". A `Code` expression is the direct equivalent to a LF `Let` expression.

```
datatype StateBehaviour
  = SB_NIX
  | SB_USE of Var
  | SB_DEF of Var * Var
```

[4] Since it is not known in advance whether a given function modifies the state, the `code` of all functions is initially generated with state arguments and state results. A later post-processing optimization removes all unnecessary function parameters.

[5] A reduction step is defined as one application of $\texttt{eval}_{\mathrm{expr}}$.

[6] The +4 was chosen because, under most function call conventions, a call involves at least a jump, a local register setup (push), a local register restore (pop), and a return jump.

[7] Using this extreme definition, syntactical means can be employed to guarantee that all primitive and function arguments are atomic. The other extreme would be to consider all expressions of finite reducibility atomic; this would only prohibit nested function applications, but allow all other kinds of nested constructs.

[8] Again, an abstract syntax is identified with its SML data type definition.

The compiler employs auxiliary functions to safely nest, compose, and locally optimize `code` expressions. To simplify these tasks, some additional bookkeeping information is provided to remember the name of the "input state" variable and the "output state" variable of each piece of `Code`.[9] A `Code` expressions either ignores the current state (`SB_NIX`), reads it (`SB_USE`), or reads and modifies it (`SB_DEF`).

```
datatype Atom =
  AVar of Var
| ALit of SV.T
```

Atomic expressions are either variable lookups or literal values. `SV.T` is the type of "semantic values", which corresponds to the domain *Obj* of the standard semantics (cf. section 3.1).

```
datatype Expr =
  Atom of Atom
| Prim of string * Atom list
| Select of Atom * (SV.T * Code) list * Code Option
| App of Var * Atom list
| Throw of int * Atom list
| Catch of int list * Code
| Func of Var list * Code
```

Expressions are either atomic, conditional, or function calls. Exception handling is modelled using explicit catch/throw forms. There are two different types of applications: `Prim` applications apply primitives (identified by strings) to argument lists; closures are applied using `App`. For the code of the called closure, `App` expressions do not belong to the LF expressions associated with closure objects, but to the CF `Code` expressions associated with the call cache entries (CCEs) maintained by the abstract interpreter. Each CCE is identified by a tag; this tag is referred to by the integer argument of the `App`.[10]

Finally, there are function expressions. These correspond to `Lambda` expressions in LF, but are restricted in that they may not contain non-global free variables.

```
datatype Decl
  = Decl of {vars : Var list, expr: Expr}
```

A declaration binds the return values of an expression to a list of variables.

CF variables are *typed* to facilitate certain post-processing optimizations, especially variable splitting/compound breakup (cf. section 8.4.5). The types form a subset of the "basic types" of the semantics. from the code-generation point of view, the types are generated automatically: each CF variable is created within a context in which the abstract value it is to hold is known; the `ac` procedure that allocates "fresh" variables generates a type annotation out of this abstract object. The types will not be mentioned in the further exposition of code generation.

8.1.3 Notation

In the following examples, an abbreviated notation for `Code` fragments is used. Instead of

$\texttt{Code\{decls=[}d_1,\ldots,d_n\texttt{],exprs=[}v_1,\ldots,v_k\texttt{],state=SB}_{\text{something}}\texttt{\}}$

a declaration list is written, followed by list of return values; the two elements are separated by a vertical line and enlosed in brackets:

```
[ d1
   ⋮
  dn
| v1, ... vk ]
```

The state behaviour is not explicitly noted.

[9] This information could also be extracted by data-flow analysis from the declarations. The explicit form was chosen both to speed up common operations, and to simplify debugging.

[10] All tags are implemented as integers, so that unique tags can cheaply be generated by counters.

8.1.4 Closures, Tuples, Combinators

LF function closures consist of three elements: a lexical variable environment, a list of typed parameters, and a "body" expression. From a given function (i.e., LF `Lambda` expression), any number of different closures can be *instantiated*. The only difference between these instances is the variable environment; parameter list and body expression are fixed.[11] It is therefore sensible to separate the run-time representation of closures into two parts: a *closure tuple* and a *combinator*.

Tuples are data-structures with a fixed number of components of arbitrary type. In the following, a convenient notation for component access will be employed: a closure tuple is created using the application of the `mk_tuple` primitive to a list of variable names, i.e. `mk_tuple(`$name_1, \ldots, name_n$`)`; if `t` is bound to a tuple, `t("`$name_i$`")` retrieves the value of $name_i$. In the actual code, the names are replaced by indices, and a lookup function `lookup_tuple(`*tuple*, *index*`)` accesses the tuple fields.

Combinators are functions that do not contain free variables; they are pure functions that can only combine their arguments. A combinator can be transformed into a sequence of machine instructions; applying the combinator corresponds to calling these instructions as as subroutine.[12] To translate a function into a combinator, its parameter list is extended by a formal parameter that will be bound to the tuple that represents the lexical environment (i.e., the closure). All occurrences of free variables in the body of the function are then mapped to `lookup_tuple` operations on this tuple parameter.

An alternative way to convert functions into combinators is to provide each needed lexical variable as an explicit parameter, instead of bundling them all together in one tuple. This is not done in `ac`, because it would greatly increase the bookkeeping effort in the AI phase of compilation; it is both easier to implement and faster to split the tuples later on, when the resident program is created and other optimizations have already minimized the code size.

8.1.5 Top-Level Bindings

Most programming languages make a distinction between top-level declarations (or *global variables*) and local declarations. Languages like C do not even allow local declarations for some entities (functions, type declarations). Even Scheme [27, 28, 62] makes a distinction between top-level bindings and other bindings (`define` and `set!` are indistinguishable at the top level).

What makes top-level objects interesting is that they can be assumed to be "fixed", i.e. they do not change their address. In pure functional languages, top-level objects are also constant, i.e. they do not change their content. In contrast, local bindings usually exist in the context of functions; their lifetime is restricted to one invocation of this function, and they have a new value and (stack) address upon each new invocation. Top-level bindings can be allocated statically at compile time; they need not be deallocated. Global variables can be treated as literals in most of the code, since they are bound to an unchanging value. It is therefore not necessary to include them in closure tuples. Program analysis and implementation is simplified to a great extent if all functions reside at the top level.

All CF function declarations generated from call cache entries are located at the top level. Also, literals that are too big to be implemented as immediate values will be represented under top-level variable bindings. Array- and list-valued literals fall under this ruling.

8.1.6 Semantics of the Code Form

The formal semantics for the Code Form is much simpler then its Lambda Form equivalent. The most important difference between them is that the Code Form semantics models exceptions by failure continuations.

[11] The types of the parameter list may also change; they are not restricted to be fixed. Apart from overloading considerations, types can be considered part from the body – they can be converted into `Lchecks`.

[12] If the combinators are in CPS form, *applying* a combinator corresponds to *jumping* to the start of its instruction sequence.

This makes it possible to specify exceptions in a "localized" fashion: in the Lambda Form semantics, each semantic function has to verify that the results of its sub-functions are "true values"; if they are not (i.e., they are exceptional), the semantic function needs to be aborted. These details were partially hidden by the "dereference" rules that were needed to cope with reference objects. Since dereferencing is made explicit in the Code Form, exception checking would surface in the semantics if modelled in the old way.[13]

Each semantic function takes two continuation functions: the "success continuation" is called with the normal result of the semantic function; the "failure continuation" is called when an exception occurs.

The second difference between the Code Form semantics and the Lambda Form semantics is the treatment of state. Code Form expressions treat the state like any other data object; coding convention guarantees that state usage is single-threaded. All computations are done on the semantic set $\texttt{Obj}_{\text{CF}}$, which includes state "values":

$\texttt{Obj}_{\text{CF}} = \texttt{Obj} \cup \texttt{State}_{\text{CF}} \cup \texttt{Functions}_{\text{CF}}$
$\texttt{State}_{\text{CF}} = \texttt{tag} \to \texttt{RefDef}$
$\texttt{Functions}_{\text{CF}} = \texttt{function(Var list, Code, Env}_{\text{CF}}\texttt{)}$
$\texttt{Env}_{\text{CF}} = \texttt{Var} \to \texttt{Obj}_{\text{CF}}$

The `Obj` and `RefDef` types are borrowed from the Lambda Form semantics (cf. chapter 3.1.5). Instead of Lambda Form `closure` objects, `function` objects are used.

The semantics of a CF program is determined by the semantic function $\texttt{eval}_{\text{CF/program}}$, which is of type

$\texttt{eval}_{\text{CF/program}}$ `: Decl list -> State`

The evaluation functions $\texttt{eval}_{\text{CF/expr}}$, $\texttt{eval}_{\text{CF/decl}}$ and $\texttt{eval}_{\text{CF/code}}$ describe the semantics of Code Form expressions, declarations and `Code` pieces, respectively. They are of type

$\texttt{eval}_{\text{CF/expr}} : \texttt{Env}_{\text{CF}} \times \texttt{Cont}_{\text{Fail}} \times (\texttt{Obj}_{\text{CF}}\ \texttt{list} \to \texttt{State}) \times \texttt{Expr} \to \texttt{State}$
$\texttt{eval}_{\text{CF/code}} : \texttt{Env}_{\text{CF}} \times \texttt{Cont}_{\text{Fail}} \times (\texttt{Obj}_{\text{CF}}\ \texttt{list} \to \texttt{State}) \times \texttt{Code} \to \texttt{State}$
$\texttt{eval}_{\text{CF/decl}} : \texttt{Env}_{\text{CF}} \times \texttt{Cont}_{\text{Fail}} \times (\texttt{Env}_{\text{CF}} \to \texttt{State}) \times \texttt{Decl} \to \texttt{State}$

All evaluation functions take a lexical environment and two continuation functions as context parameters. All failure continuation are of the same type

$\texttt{Cont}_{\text{fail}} = \texttt{tag} \times \texttt{Obj}_{\text{CF}}\ \texttt{list} \to \texttt{State}$

Success continuations are of type "$\texttt{x} \to \texttt{State}$", where `x` is the result of the evaluation function; the success continuation is called with this result when no exception is raised. Expressions and `Code` pieces evaluate to object lists, while declarations evaluate to environments.

Since the exceptions have been isolated, all evaluation functions return (or pass on) valid object lists instead of *Results*.[14]

In the following, semantic environments are named `E`, success continuations are denoted `K`, and failure continuations are named `F`.

8.1.6.1 Programs

A CF program is a list of declarations, one of which must define a function called "`init`".

[13] An alternative semantic model that was considered employed the notion of explicit stacks: instead of continuations, stack positions would then be recorded; raising an exception would correspond to "jumping up the stack". Such a model reflects actual implementation practice, but does not simplify formal reasoning. Like the continuation-passing semantics that is now employed, a stack model needs to introduce "labels" (i.e. continuation functions) to put on the control stack, so the semantics wouldn't be any smaller.

[14] If they return at all, that is. The Code Form cannot guarantee termination, even though the abstract interpreter must terminate to generate the Code Form program.

```
evalCF/program ([decl1,...,decln]) =
    evalCF/expr(Eglobal, Failglobal, Termglobal, App("init", closureinit, stateinit)
where
    Eglobal = Efuncts + Estatic
    Efuncts = { (name,function(params,code,Eglobal))
                | ∃ i: decli = Decl({vars=[name],expr=Func(params,code)}) }
    Estatic = { (name,value)
                | ∃ i: decli = Decl({vars=[name],expr=ALit(value)}) }
    Failglobal = λ args . error("uncatched exception")
    Termglobal = λ state . state
    closureinit = Efuncts("init")
    stateinit = ∅
```

The global (or static) values must be literals such as tables. The static values and the functions comprise the global environment; each function has the global environment as its "closure". The actual program is run by invoking the "`init`" function with the initial state (an empty reference definition set). Upon successful termination, the $\texttt{Term}_{\text{global}}$ continuation will be called with the result state.

8.1.6.2 Atoms

The definition of $\texttt{eval}_{\text{CF/expr}}$ employs an auxiliary function $\texttt{eval}_{\text{CF/atom}}$ for atomic sub-expressions:

```
evalCF/expr(E,K,F,Atom(a)) = K(evalCF/atom(E,a))
evalCF/atom(E,AVar(var)) = E(var)
evalCF/atom(E,ALit(lit)) = lit
```

There is no need to pass continuation functions to $\texttt{eval}_{\text{CF/atom}}$, since atomic expressions cannot raise exceptions. It can be syntactically guaranteed that that all variables are bound; therefore the variable lookup `E(var)` cannot fail.

8.1.6.3 Primitives

The behaviour of primitives shall not be defined here (i.e., $\texttt{appy}_{\text{CF/prim}}$ is left unspecified), it suffices to say that all primitives are pure functions: it is not possible to access the call cache or modify global state using a primitive. The essential primitives (without which execution would be impossible) are described in the examples, when they occur for the first time.

Most of the other primitives have the same semantics as under $\texttt{apply}_{\text{primitive/strict}}$ (cf. chapter 3.5.3).

```
evalCF/expr(E,K,F,Prim(s,arg1,...,argn)) =
    K(applyCF/prim(s,evalCF/atom(E,arg1),...,evalCF/atom(E,argn)))
```

Note that there is no order of argument evaluation implied. Atomic expressions cannot generate side effects; therefore the order of their evaluation need not be specified.

Note also that primitives cannot fail; there is no possibility to raise an exception from "within" a primitive.[15]

[15] This begs the question of how to model arithmetic primitives that can raise exceptions, such as division (which can raise exceptions on underflow, overflow, or division-by-zero). In the ALDiSP arithmetic standard library, such operations are implemented by explicit tests. To keep with the division example, each use of the primitive division operator is preceded by tests for possible underflows, overflows, and zero-ness of the second argument. In any such case, an exception is raised excplicitly. The reconstruction process will re-create these tests in the Code Form(if the abstract interpreter has not been able to remove them as dead code). If the target machine supports an operation that performs a division and generates extra "exception flags", the back-end code generator (cf. chapter 9) has to detect the opportunity to utilize this operation to implement the more general approach. This is an instance of an *idiom detection* technique.

8.1.6.4 Conditionals

The Code Form supports only one, very general, conditional. It resembles the "`Lselect`" of the Lambda Form:

```
eval_CF/expr(E,K,F,Select(a,[(key_1,code_1),...,(key_n,code_n)],code_default)) =
    if ∃ i : key_i = key_val then eval_CF/code(E,K,F,code_i)
    else if code_default ≠ NONE then eval_CF/code(E,K,F,code_default)
                                 else error()
where
    key_val = eval_CF/atom(E,a)
```

The default is optional so that no "dummy code" is needed if the compiler knows that the `key`s are exhaustive.[16]

No ordering is implied in the `Select` form; if two keys are identical, either can be chosen.[17]

8.1.6.5 Function Application

Function applications follow some conventions. The first argument is the function's closure, the last argument is the state variable (if the function takes a state argument). In between are the "real" arguments. If the function returns an altered state, it will be the last element in the return value list.

The closures have the same form as in the Lambda Form semantics; a closure consists of a lexical environment and a Lambda Form expression. This expression is ignored; instead, the integer index that is part of the `App` node is used to look up a `Code` expression in the call cache.[18]

The closure need not be analyzed by the semantics at all; at the Code Form level, all functions are combinators (i.e., have no state). The semantics only binds the arguments to the formal parameters; this gives the environment in which the function's body is evaluated.

```
eval_CF/expr(E,K,F,App(name,arg_0,...arg_n)) =
   let
      func([v_1,...,v_n],code_func,E_global) = E(name)
      ∀ i ∈ 1..n : val_i = eval_CF/atom(E,arg_i)
   in
      eval_CF/code(E_global+[(v_1,val_1),...,(v_n,val_n)],K,F,code_func)
```

8.1.6.6 Throwing and Catching Exceptions

Throwing an exception is performed by calling the failure continuation with the supplied arguments:

```
eval_CF/expr(E,K,F,Throw(tag,arg_1,...,arg_n)) =
    F(tag,eval_CF/atom(E,arg_1),...,eval_CF/atom(E,arg_n))
```

The tag identifies the exception that was thrown. Catching exceptions involves the creation of a new failure continuation. When called (by a `Throw`), this function tests the tag that was thrown; if it is not part of the list of tags to be catched, the exception is re-thrown.:

```
eval_CF/expr(E,K,F,Catch([tag_1,...,tag_k],code)) =
    eval_CF/code(E,K,F',code)
where
    F' = λ(tag,[v_1,...,v_n]).
     if ∃ i: tag_i = tag then K([v_1,...,v_n])
                              else F(tag,[v_1,...,v_n])
```

[16] In SML, the type `'a Default` is defined as `NONE | SOME('a)`. The semantics is written in a meta-SML, in which the SOME can be dropped.

[17] In practice, this should never happen.

[18] For all practical purposes, the `closure()` type can be remodelled by a tagged tuple, i.e. `closure_CF(tag_c,tag_l,env)`.

8.1.6.7 Evaluating Declarations

Evaluating a declaration involves evaluating the expression of the declaration with a new success continuation.

```
eval_CF/decl(E,K,F,Decl({vars=[var_1,...,var_n],expr=e})) =
    eval_CF/expr(E,K',F,e)
where
    K' = λ([v_1,...,v_n]).
              K(E + [var_1/v1,...,var_n/vn ])
```

When called with the results, the success continuation will update the old environment with the new bindings, and call its continuation with this new environment.

8.1.6.8 Evaluating `Code` expressions

A `Code` expression is evaluated in two stages: first, the declarations are sequentially evaluated, resulting in a final environment. Then, using this environment, the atoms are evaluated that make up the return values.

```
eval_CF/code(E,K,F,[decl_1 ... decl_n | atom_1 ... atom_k]) =
    if n=0 then
        K([eval_CF/atom(E,atom_1),...,eval_CF/atom(E,atom_k)])
    else
        eval_CF/decl(E,K',F,decl_1)
where
    K' = λ(E').
              eval_CF/code(E',K,F,[decl_2 ... decl_n | atom_1 ... atom_k])
```

This concludes the semantics of Code Form expressions.

8.2 Relation between Abstract Results and Code Attributes

This section discusses how and why each abstract *Result* gets its `code` attribute, and what the semantic link between them is.

The approach to code representation and generation that is taken in the development of `ac` is an evolutionary one. Beginning with a standard semantics, an abstract semantics is designed and implemented in the form of an interpreter. Code generation is then added to this interpreter. The basic restriction on code representation and code generation is therefore the fact that the abstract interpreter does not know about them!

To achieve this goal, i.e. to work within the boundaries of the abstract semantics, attributes are attached to the abstract semantic values. Attributes must be orthogonal to the abstract value model; the evaluation model totally ignores them. The attributes contain not only the code fragments themselves, but also all information needed to keep the generated code pieces coherent.

During the abstract evaluation of the original LF program, abstract *Result*s are generated. It was already mentioned in section 6.2.1 that any domain of abstract values can be extended with arbitrary orthogonal attributes. In the case of the `ac` abstract *Result*s, there is such an attribute called `code`, which carries a value of type `Code`. For a given abstract *Result*, this attribute represents the code that has to be executed at run time to generate the concrete result. A second attribute `atom` (which has an `Atom` as its value) is attached to each abstract `Object`.

8.2.1 The Treatment of State

The biggest conceptual problem in generating the `code` attribute is the treatment of state. In the standard semantics, most evaluation functions return *Result*s that consist of an object and a state. Often, the state

will be identical to the state that was passed to the semantic function as part of the evaluation context; sometimes, the state has changed. For the purpose of this discussion, a "state" is a set of reference value definitions, similar to an environment or a tuple of values. There are only a few possible transformations that can change a state during evaluation: a new reference may be added (by a suspension or promise creation), or an existing reference may be updated (by forcing a promise). Removing references and changing the status of suspensions is performed by the scheduler.

The `code` attribute is intended to model the computation process necessary to produce a run-time *Result*. An alternative approach is to attach a `code` attribute to each *value* (in our semantics, to each *Obj*). As a major consequence of the per-*Result* attribution, only one code fragment is generated by each abstract semantic function.[19]

The "current state" is treated as a variable that denotes a set of tagged values. A reference can be accessed via special primitives `_deref` and `_update`. `_deref(`*state*, *ref*`)` returns the value associated with *ref*, where *ref* is a currently defined element of the set of reference tags, and *state* is the "current state" variable; `_update(`*state*, *ref*, *val*`)` creates or updates a reference value.[20] The result of `_update` is a new state.

The state is supposed to be single-threaded; it is illegal to `_update` a state twice, or to `_deref` a state that has already been updated. If a `Code` fragment does not contain `_update` applications and does not call side-effecting functions, it is side-effect free.

8.2.2 `atom`: a Code Attribute for *Objs*

The `atom` attribute was introduced to solve the problem of *identifying* values. In the context of abstract interpretation, it is possible to generate values that are equal, but not identical. For example, if `a` is bound to `anInteger`, `a+1` and `a-1` are both `anInteger`, but they are different. When generating code, it is necessary to know the expression a given object is bound to. Therefore, a second Code attribute has to be introduced: the `atom`. Each *Obj* has an `atom` attribute which describes the atomic expression that will denote the *Obj* at run-time. The standard and abstract operations on *Obj*s do not change; they ignore the `atom`.[21]

The `atom` attribute is only useful when seen in the context of a `code` fragment. One can visualize the relation between `atom` and `code` as that of a pointer to a node in a directed graph: most `atom`s will be variable lookups, which refer to their definitions in the `code`. The `code` can be viewed as a graph, since the declaration list of a `Code` fragment can be arbitrarily reordered, as long as the def-use relationships between the definitions are fulfilled. Since there are no implicit def-use relationships between declarations, the declaration list can be assumed to be unordered.

8.3 Example: Straight-Line Code

In the following, code generation shall be traced step-by-step for the following ALDiSP program fragment:

```
let
  x = delay(2*y)
in
  (y + x) + x
end
```

[19] This design decision was not taken lightly. It took me quite some time to recognize that it is the *Results* that have state and must therefore be annotated with the `code` attribute, not the *Obj*ects! Most published literature annotes the values – because literature does not describe partial evaluators that have to cope with state, exceptions, and multiple results!

[20] There is no practical difference between creating and updating a reference. There is only a fixed set of reference tags; no new references can be generated at run-time. The storage for reference definitions is therefore allocated statically at compile time.

[21] The precision of the abstract equality test can be improved, since objects with identical `atom` must be identical.

This example, while rather tiny, exercises code generation for literals, primitive applications, a function application, tuple creation, all operations related to references, and the handling of declarations.

We assume that `y` is a variable defined in some outer scope and bound to a non-literal value. This value has an `atom` attribute, which shall be called $\texttt{atom}_y$. Let the initial state be bound to the CF variable `state_initial`.

The ALDiSP-to-Lambda Form transformation (cf. chapter 4.4.4) will have generated an LF expression[22]

```
let
  x = _delay(lambda_tag(){2*y})
in
  (y + x) + x
end
```

Evaluation starts with $\texttt{eval}_{\text{expr/Let}}$,[23] which calls $\texttt{eval}_{\text{decl}}$ to evaluate the declaration of `x`. $\texttt{eval}_{\text{decl}}$ calls $\texttt{eval}_{\text{expr}}$ on the `_delay` application.[24] $\texttt{eval}_{\text{expr/Lapp}}$ calls $\texttt{eval}_{\text{exprs}}$ to evaluate `_delay` and its one argument, the `Lambda` expression. $\texttt{eval}_{\text{expr}}$`(Lit(_delay))` gives rise to the first *Result*:

```
[ | _delay ]
```

The `atom` attribute of the returned `_delay` object is the CF atom literal `ALit(_delay)` (or simply, `_delay`).

$\texttt{eval}_{\text{expr/Lambda}}$ gives rise to a closure containing the binding of its free variables, `y` and `*`. The semantics mentions an auxiliary function `free` that determines what variables are closed over. `free` does not know that `y` is a global function, and will include it in the closure. For code generation purposes, a different function `free_codegen` determines the variables bound in the `code` expression. Since the `Lambda` has no arguments, the $\texttt{eval}_{\text{exprs}}$ application that evaluates the argument types calls its continuation immediately, and creates no code. The `code` associated with the *Result* is `mk_tuple(`$\texttt{atom}_y$`)`; since tuple creation is not atomic, a variable $\texttt{tmp}_1$ is introduced. The full `code` is

```
[ tmp_1 = mk_tuple(atom_y)
| tmp_1 ]
```

The *Closure* object has $\texttt{tmp}_1$ as its `atom` attribute. Note that there is no connection between this code and the code generated earlier for the `_delay` primitive. The different code attributes have to be combined later on.

Now $\texttt{eval}_{\text{exprs}}$ finishes and calls its continuation function, which points back into $\texttt{eval}_{\text{expr/Lapp}}$. Here, $\texttt{deref}_{\text{objs}}$ is first applied to the function argument. Since `_delay` is not a reference, $\texttt{eval}_{\text{expr}}$ calls $\texttt{apply}_{\text{mappable}}$ to do the actual application.

$\texttt{apply}_{\text{mappable}}$`(_delay,...)` calls $\texttt{test}_{\text{apply}}$ to determine whether the application has to be mapped. $\texttt{test}_{\text{apply}}$ decides that `_delay` is a primitive and calls $\texttt{test}_{\text{apply primitive}}$, which returns `true` (since all primitives are assumed to be defined on all arguments). $\texttt{apply}_{\text{mappable}}$ then calls `apply`, which in turn calls $\texttt{apply}_{\text{primitive}}$, which locates `_delay` in its list of non-strict primitives and calls $\texttt{apply}_{\text{primitive/nonstrict}}$. Finally, the semantic action associated with `_delay` is performed: a new reference is allocated in the current state; an `Uprom` (cf. chapters 3.1.5 and 7.1) containing the closure is stored in that reference; the tag that addresses the

[22] Without syntactic sugar, the transformed expression looks like this:

```
Let(Lapp([Lprim(+),
          Lapp([Lprim(+),Lvar(y),Lvar(x)]),
          Lvar(x)]),
    Ldecl(true, x, Lapp([Lprim(_delay,
                         Lambda(tag,[],Lapp([Lprim(*),Lit(2),Lvar(y)]))]))))
```

This is hard to read and adds no relevant information. In all following examples, a hybrid syntax will therefore be employed.

[23] As a convention, large semantic functions like $\texttt{eval}_{\text{expr}}$ are grouped into "rule sets" relating to the kind of expression they take as their first argument

[24] $\texttt{eval}_{\text{decl}}$ calls $\texttt{eval}_{\text{expr}}$ indirectly via $\texttt{eval}_{\text{exprs}}$. Such small evaluation steps and indirection will often be omitted in the future discussion.

reference in the state is returned as *Result*. The tag (in the following called $\texttt{tag}_{\text{delay}}$) is a literal and has as itself as `atom`.

For each primitive function application, a corresponding `Code` object is created. The `_delay` function emits code that consists of an `_update` primitive. The result of the `_update` is a new state, for which a new variable has to be introduced:

```
[ state_new= _update(state_initial,tag_delay,tmp_1)
| tmp_1 ]
```

Note the missing declaration for $\texttt{tmp}_1$, which is currently not visible. The `_update` reflects the fact that the closure ($\texttt{tmp}_1$) has been stored in the state; it does not make explicit that it is stored in an `Uprom`. This information is not needed at the code generation level. Likewise, the allocation of a reference tag is not made explicit, because the tag is not an entity that changes at run-time.

$\texttt{apply}_{\text{primitive/nonstrict}}$ returns its result (via $\texttt{apply}_{\text{primitive}}$, `apply`, and $\texttt{apply}_{\text{mappable}}$) to the second continuation of $\texttt{eval}_{\text{expr/Lapp}}$. This continuation returns to $\texttt{deref}_{\text{objs}}$, which returns to the first continuation of $\texttt{eval}_{\text{expr/Lapp}}$, which returns to $\texttt{eval}_{\text{exprs}}$.

At this point, code fragments are combined for the first time. The necessity for this can be seen by looking at the type of $\texttt{eval}_{\text{exprs}}$: it evaluates expressions to results, but calls its continuation function with a list of objects. Since results carry a code attribute, which objects don't have, there is a loss of information in passing along the objects to the continuation. When the continuation returns, this information has to be added again to the final result (otherwise, the code would be lost!).

This is done by concatenating the declaration lists of the code attributes of the *Result*s that were computed, and prefixing them to the result that is passed back. $\texttt{eval}_{\text{exprs}}$ calls itself recursively for each element in the expression list it evaluates; each of these calls strips the *Obj* from one *Result*, and puts the *Result*'s declaration list back upon return. In our example, $\texttt{eval}_{\text{exprs}}$ has evaluated two expressions, of which the first (`_delay`) has an empty declaration list; the second (the `Lambda`) contains one declaration. $\texttt{eval}_{\text{exprs}}$ returns its *Result* (still the reference tag that denotes the `Ususp`) with the code attribute

```
[ tmp_1 = mk_tuple(atom_y)
  state_new= _update(state_initial,tag_delay,tmp_1)
| tmp_1 ]
```

to $\texttt{eval}_{\text{expr/Lapp}}$, which returns to the $\texttt{eval}_{\text{exprs}}$ that was called from $\texttt{eval}_{\text{decl}}$. Again, the object is stripped from the result and passed along to the first continuation of $\texttt{eval}_{\text{decl}}$. This continuation creates a singleton environment in which the variable `x` (*not* the `atom` of the object to be bound, but the ALDiSP-level variable!) is bound to the $\texttt{tmp}_1$ object. $\texttt{eval}_{\text{decl}}$ then calls that continuation that was passed down to it, which is the first continuation $\texttt{eval}_{\text{expr/Let}}$.

We now start to evaluate the `(y+x)+x` expression by calling $\texttt{eval}_{\text{expr}}$ with it.

$\texttt{eval}_{\text{expr/Lapp}}$ calls $\texttt{eval}_{\text{exprs}}$ for its three arguments (`+`, `(y+x)`, and `x`). $\texttt{eval}_{\text{exprs}}$ calls $\texttt{eval}_{\text{expr/Lit}}$, which returns a *Result* that denotes the primitive `+`.[25] $\texttt{eval}_{\text{exprs}}$ strips the *Obj* from this *Result*, and recurs. The next incarnation of $\texttt{eval}_{\text{exprs}}$ invokes $\texttt{eval}_{\text{expr/Lapp}}$ (on the `y+x` expression), which calls $\texttt{eval}_{\text{exprs}}$ with the function/argument list (`[+,y,x]`). The second `+` evaluates identical to the first.

When applied to `y`, $\texttt{eval}_{\text{expr/Lvar}}$ accesses the value of `y` and generates code for it. This code is simply `[|y]` (we assume that the object to which `y` is bound has `y` as its atom). The `atom` attribute of the *Obj* is therefore `y`.

Evaluating `Lvar(x)` works identical to `Lvar(y)`, and returns the reference tag object that denotes the `Ususp` that was constructed earlier on. $\texttt{eval}_{\text{exprs}}$ calls the continuation of $\texttt{eval}_{\text{expr/Lapp}}$, which calls $\texttt{deref}_{\text{objs}}$ with the function (`+`), which calls $\texttt{apply}_{\text{mappable}}$, which calls `apply`, which calls $\texttt{apply}_{\text{primitive}}$.

[25] In real life, `+` is overloaded; for pedagogical purposes, this example sticks to primitives. Following the execution trace through the convoluted hierarchy of `apply` functions is bad enough.

Since + is a strict primitive, $\texttt{deref}_{objs}$ is called with the values of y and x. Since the value of x is refers to a Uprom, $\texttt{deref}_{objs}$ calls $\texttt{eval}_{thunk}$ with the closure of the Uprom and the current state. Just to be sure, $\texttt{eval}_{thunk}$ first pipes the closure through another instance of $\texttt{deref}_{objs}$ (this is needed to cope with nested references). A CF declaration is generated to model the state lookup:

```
[ tmp2 = _deref(state_new,tag_delay)
| tmp2 ]
```

Then it calls apply with an empty argument list. Without any testing, apply calls $\texttt{apply}_{closure}$. There is no call cache entry that corresponds to the closure, therefore a new one is introduced. The closure's environment is extended with the (empty) parameter binding list, and $\texttt{eval}_{expr}$ is applied to the body expression, y*2. Skipping over evaluation steps (that would be similar to the evaluation of the _delay), the result is an abstract value with code

```
[ tmp_y = lookup_tuple(tmp_tuple,"y")
  tmp3 = tmp_y*2
| tmp3,state_input ]
```

Parameters $\texttt{tmp}_{tuple}$ and $\texttt{state}_{input}$ are a newly generated upon each closure application and introduced into the parameter list at the first and last position, respectively. It is assumed that each function can modify the state, therefore a state parameter and return value is always generated. Also, it is assumed that all function take a closure argumet. If a function turns out to be a combinator, or to not access any of its lexically bound variables, the tuple parameter is removed by a later optimization, as is the state parameter/argument when the function is side-effect free.

This *Result* and its code is memoized in the call cache entry; $\texttt{apply}_{closure}$ generates a new code fragment that represents the call:

```
[ tmp4, state_newer = App(tag_cce,tmp4,state_new)
| tmp4 ]
```

The $\texttt{tag}_{cce}$ is the call cache entry tag of the application. This is later needed to retrieve the code. It is not possible to inline the code yet, since recursion might be involved.

While $\texttt{apply}_{closure}$ knows that no state change has happened, it has to create a new state variable and bind it. The same function might, after all, change the state when called in another context!

$\texttt{apply}_{closure}$ also introduces a new variable $\texttt{tmp}_4$ to hold the result of the application, and changes the atom of the result object to $\texttt{tmp}_4$. The state is passed into the application, and the right side of the application contains a multiple definition for both a fresh state variable and the result. Control returns from $\texttt{apply}_{closure}$ to $\texttt{apply}_{thunk}$, which pipes the result through $\texttt{norm}_{result}$ to asserts that it is not an exception.[26]

$\texttt{norm}_{result}$ returns the *Result* to the $\texttt{deref}_{objs}$ that had initiated the Uprom evaluation. The Uprom is replaced by an EvRef in the current state. The state lookup code is added at the beginning, and the state update code is added at the end of the Uprom code:

```
[ tmp2 = _deref(state_new,tag_delay)
  tmp4,state_newer = App(tag_cce,tmp2,state_new)
  state_newest = _update(state_newer,tag_delay,tmp4)
| tmp4 ]
```

$\texttt{norm}_{result}$ then calls its continuation with the normalized *Obj* list, whose atoms are [y,tmp4] . That continuation proceeds with applying + to the normalized arguments. apply and $eval_{exprs}$ return with the code

```
[ tmp5 = atom_y + tmp4
| tmp5 ]
```

[26] Here, a small change to the semantics is in order: $\texttt{norm}_{result}$ returns a (*Obj*, *State*) tuple instead of a result, i.e. it strips a *Result*. It is not possible to re-attach the stripped declarations to the atom of the *Obj*, since $\texttt{norm}_{result}$ does not take a continuation. By changing the range of $\texttt{norm}_{result}$ to *Result*, this problem disappears. $\texttt{norm}_{result}$ occurs three times in the semantics[27], each of these is easily "repaired". This change in the semantics does not change its behaviour. It is still regrettable that the semantics had to be changed to accomodate the code generator.

The second pending invocation of $\texttt{deref}_{\text{objs}}$ attaches the dereference/apply/updating code to the result and returns; the first pending invocation passes the result directly back.

This finishes the `eval_expr/Lapp` of the inner addition. Control returns to the outer addition's $eval_{exprs}$, which encounters the variable `x` and evaluates it to its reference value. Skipping the next few steps, evaluation arrives at the next `apply_primitive/strict`. Again, $\texttt{deref}_{\text{objs}}$ is called with two arguments, of which the second is a reference with `x` as its `atom`. This time however, the reference is bound to an *EvRef*. The code that models the lookup is generated:

```
[ tmp6 = _deref(state_newest,tag_delay)
| tmp6]
```

and the *Obj* that was stored in the state gets $\texttt{tmp}_6$ as its new `atom` (the old one, $\texttt{tmp}_4$, is still visible in the current context, but the code generator can't know this). $\texttt{deref}_{\text{objs}}$ calls its continuation, which finishes the application and generates the `code` fragment

```
[ tmp7 = tmp5 + tmp6
| tmp7 ]
```

Control returns to $\texttt{deref}_{\text{objs}}$, which attaches the deref preamble, and returns the `code`

```
[ tmp6 = _deref(state_newest,tag_delay)
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

to the enclosing second continuation of $\texttt{eval}_{\text{expr/Lapp}}$. This returns through $\texttt{deref}_{\text{objs}}$, which attaches the force/apply/update `code`, generating

```
[ tmp2 = _deref(state_new,tag_delay)
  tmp4,state_newer = App(tag_cce,tmp2,state_new)
  state_newest = _update(state_newer,tag_delay,tmp4)
  tmp5 = atom_y + tmp4
  tmp6 = _deref(state_newest,tag_delay)
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

Finally, control returns to the continuation of $\texttt{eval}_{\text{exprs/Let}}$, which returns (through $\texttt{eval}_{\text{decl/Ldecl}}$) to $\texttt{eval}_{\text{exprs}}$, where the missing declaration for $\texttt{tmp}_1$ is prefixed. Evaluation finishes with a `code` attribute that is

```
[ tmp1 = mk_tuple(atom_y)
  state_new= _update(state_initial,tag_delay,tmp1)
  tmp2 = _deref(state_new,tag_delay)
  tmp4,state_newer = App(tag_cce,tmp2,state_new)
  state_newest = _update(state_newer,tag_delay,tmp4)
  tmp5 = atom_y + tmp4
  tmp6 = _deref(state_newest,tag_delay)
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

This concludes the first example. The code that was generated is somewhat inefficient, giving ample motivation for the optimizations that are presented next.

8.4 Inter-Function Optimizations

The first group of optimizations that shall be considered is concerned with optimizing function calls (i.e., `Apps`), by either modifying a function's call/return interface, or by outright eliminating the application.

8.4.1 *Inlining*

Inlining of CF function applications is an obvious optimization. The decision to inline a specific function application is deferred to the reconstruction phase. Too much inlining can lead to code explosion, accompanied by an increase in compilation time that is more than linear in the code size. There exists no "optimal" inlining strategy, since there is no direct space/time trade-off: once a function application has been inlined (which will have increased the code size), many other optimizations (which might reduce the final code size) become possible. The heuristics that decide upon inlining in `ac` are fairly obvious:

- All applications to *trivial* functions are inlined. An application is deemed trivial if the inlined code is of the same size or smaller than the code which it replaces, i.e. the function application.[28] The "size" of a piece of code is measured by its reducibility.[29]
- If a function is applied only once, it is inlined.
- If a function is applied only once in a given execution path, and if its size is less than 50, it is inlined.[30]
- If a function is applied in the lexical context of its closure tuple's creation, and if its size is less than 40,[31] it should be inlined. This avoids unneccessary tuple creation and lookup operations.

The mechanism of inlining is trivial: given a call cache entry

$\texttt{CallCache(tag}_f\texttt{) = ([decl}_1\texttt{;...;decl}_j \texttt{ | expr}_1\texttt{,...expr}_k\texttt{], [p}_1\texttt{,...,p}_n\texttt{])}$

and a function application

$\texttt{res}_1\texttt{,...,res}_k \texttt{ = App(tag}_f\texttt{,a}_1\texttt{,...a}_n\texttt{)}$

Then the application can be inlined as

$\texttt{p}_1 \texttt{ = a}_1$

$\vdots$

$\texttt{p}_n \texttt{ = a}_n$

$\texttt{decl}_1$

$\vdots$

$\texttt{decl}_j$

$\texttt{res}_1 \texttt{ = expr}_1$

$\vdots$

$\texttt{res}_k \texttt{ = expr}_k$

The inlining mechanism does not substitute an $\texttt{a}_i$ for each occurrence of a $\texttt{p}_i$. Eliminating the $\texttt{p}_i\texttt{=a}_i$ declarations is considered to be a separate optimization, implemented as a post-pass.

As part of the inlining, all variable names that are local to the residual code (this includes the parameter names) are mapped to new unique names. This guarantees that no name-clashes can occur when a function is inlined more than once in the same `code` expression.

Assuming that the `App` in the example is inlined, the code will be:

[28] That is, a function is trivial if its body has a reducibility < 4.

[29] The reducibility does not necessarily correspond to the final assembly code size. For example, replacing a variable with a literal value might largely increase the size of a program if the literal is not atomic (e.g., if it is an array). The current back-end folds multiple copies of the same literal into on representation, so that this specific effect can be ignored in the reconstruction phase.

[30] The number "50" is an obviously arbitrary heuristic; it can be changed at any time.

[31] Again, an arbitrary heuristic.

```
[ tmp_1 = mk_tuple(atom_y)
  state_new= _update(state_initial,tag_delay,tmp_1)
  tmp_2 = _deref(state_new,tag_delay)
  state_input' = state_new
  tmp_tuple' = tmp_2
  tmp_y' = tmp_tuple'("y")
  tmp_3' = tmp_y'*2
  state_newer = state_input'
  tmp_4 = tmp_3'
  state_newest = _update(state_newer,tag_delay,tmp_4)
  tmp_5 = atom_y + tmp_4
  tmp_6 = _deref(state_newest,tag_delay)
  tmp_7 = tmp_5 + tmp_6
| tmp_7 ]
```

The renaming of variables is indicated by the tick symbol, i.e. $\texttt{tmp}_2$ has been renamed into $\texttt{tmp}_2$'.

The example presumes that the transformation which introduces the explicit tuple closure parameter (cf. section 8.1.4) has already been applied to both the example `code` and the call cache entry. The $\texttt{tmp}_{\text{tuple}}$ variable holds the tuple; $\texttt{tmp}_{\text{y}}$ holds the value from `y` that is extracted from the tuple.

8.4.2 Grouping

A problem closely related to inlining is the "grouping" of functions in the call cache. Given a function f that is called 15 times during the execution of the program, how many instances of the function shall exist in the final code? In the call cache, there will exist a distinct residual code attribute for each of these calls.[32] It is often possible to share one code representation for a group of these residuals. This does not improve the code "quality" (i.e., it does not increase the run time speed), but it decreases the global code size. Since code explosion is one of the gravest concerns of partial evaluation, the grouping of residuals is very important. Grouping can be much improved if small changes to the residuals are allowed.

Again, some heuristics are used:

- If a subset of the applications generate the same (modulo renaming) residual code, this subset can be trivially grouped.
- If there is a set X of applications that differ only in the values of a literal c, which has the same basic type in all residuals in X, then c can be replaced by a variable, which is added as an extra parameter to the function interface. No efficiency is lost, since there exists no possibility for value-dependent optimizations – all such optimizations have already been performed by the abstract interpreter.

8.4.3 Tabulation and Caching

Functions can be represented by tables when their domain is known at compile time and they perform no side effects. The most natural candidates for such a representation are functions of one or two arguments, when the arguments are known to be restricted to a small enumerable type. Such functions correspond to vectors and matrices. Tabulation is only useful if the tabulated function perform a non-trivial computation. The time needed to compute a functions must be compared with the size of the final table and the time for a lookup operation. In the case of connected argument sets (ranges of integers with no "holes"), cost of lookup can be approximated as one multiplication and one addition for each argument, plus a memory lookup.

Heuristics guiding tabulation are

[32] The different calls might have been mapped together if they have exactly the same arguments, but even so simple a difference as that between `f(anInt,1)` and `f(anInt,2)` might cause the creation of different entries.

- If the index set has more than 256 elements, no tabulation should take place.
- If the computed function is k-reducible, with $k \leq 7$, no tabulation takes place. "7" is of course an arbitrary choice; reducibility might be weighted for the different costs of primitive functions.
- If the index set is not a numeric type, a lookup function has to be generated that either performs a structural search or employs some kind of hashing. This increases the cost of tabulation, to a logarithmic number of comparison operations and `Lcond/Lselect` nodes, if timing is considered, and is linear in code size.[33]
- If the index set is very small (≤ 4), tabulation should always be done.

A function is tabulated by replacing its body with a `Select` expression. Each table entry corresponds to one arm of the `Select`; there is no default case. This table representation gives the back-end the most flexibility in representing the table.[34]

8.4.4 *Elimination of Parameters and Return Values*

Parameter elimination removes parameters of a function definition if these are never used in its body. Such parameters may occur due to function specialization, i.e. whenever a parameter is only used in a conditional branch that was removed because the condition was known. To be correct, all calling sites of the function must be identified, and the arguments that match the eliminated parameter must be removed. This may introduce new dead code in the calling sites.

If a function returns a value that is statically known, either because it is a parameter to the function or because it is a literal, said value can be eliminated from the function's expression list; the variables that are bound to this parameter at the calling sites have to be re-bound to the statically known value. This optimization specifically addresses state variables that are passed through functions unchanged. As a side effect, this may remove the last use of a parameter, thus making it eligible for parameter elimination. Return values can also be removed if they are never used.

8.4.5 *Variable Splitting*

If a function accepts a parameter that is bound to a compound value (i.e., a tuple or the state), and this function does "consume" the parameter without passing it on, and if only a subset of the element of the compound are accessed, then it is an optimization to break the compound into its parts and transform the compound parameter into a set of part parameters. The generalized application of this transformation amounts to "splitting" variables that hold compound values into groups of variables, each of which holds a sub-value of an atomic type.

This optimization is targetted at functions that access reference values from a state, but do not change the state. It also serves to break up closure tuples, thus avoiding the `mk_tuple` creation. If the lifetimes of compound parameters differ, breaking them up may optimize their allocation behaviour, since a compound has the lifetime of its longest-living component.

Compound breakup for state variables does not gain as much from a variable-passing point of view, since the back-end will implement reference variables in terms of imperative variables (this is made possible by the single-threaded-ness of the state). However, the "function sharing" optimization becomes feasible.

---

[33] In any case, generating such code is non-trivial and possibly not worth the effort. ac does not consider generating code for non-integer index sets.

[34] The alternative would be a "direct" implementation as a global array literal.

8.5 Basic Optimizations

The code that was generated for the example suffers from obvious inefficiencies. This section shows some analysis and transformation steps that will remove most of them. Most of these steps are part of the "real" reconstruction phase; they cannot be performed ere the full program has been interpreted. These optimizations are called "basic" because they work on the level of the `Code` expression, i.e. they do not consider the interface between functions.[35]

8.5.1 Variable Propagation

The example code shows many occurrences of variable renamings, i.e. definitions of the form `x = y`. Such a definition can be transformed by changing every succeeding use of `x` into a use of `y`. Due to this change, `x` will become a "dead" variable, and its definition dead code. Variable propagation may have to be iterated, since successive renamings of the same value have to be considered.

In the inlined example, variable propagation applies to the set {$state_{input}$', tmp_{tuple}', $state_{newer}$, tmp_4}. The variable $state_{newer}$ gives an example for iterated renaming. After propagation, the code is

```
[ tmp1 = mk_tuple(atomy)
  statenew= _update(stateinitial,tagdelay,tmp1)
  tmp2 = _deref(statenew,tagdelay)
  stateinput' = statenew
  tmptuple' = tmp2
  tmpy' = tmp2("y")
  tmp3' = tmpy'*2
  statenewer = statenew
  tmp4 = tmp3'
  statenewest = _update(statenew,tagdelay,tmp3')
  tmp5 = atomy + tmp3'
  tmp6 = _deref(statenewest,tagdelay)
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

8.5.2 Dead-Code Removal

For each Code Form code fragment, the "output" consists of the set of expressions plus the "last" state. Any variable binding that does not contribute to the output is not "alive", it can be eliminated as dead code. The sole exception to this is the state variable, which is an considered an implicit output value.

The most simple of data-flow analyses can be performed to compute the set of living variables: if a variable occurs in the output list, it is alive. If a variable is part of an expression that defines a alive variable, it is alive itself.

For the example in its inlined and forwarded form, dead-code analysis shows that the variable set {tmp_{tuple}, tmp_4, $state_{input}$', $state_{newer}$} is not alive any more; therefore their defining declaration can be removed, resulting in the code

[35] "Basic" does *not* refer to "basic blocks".

```
[ tmp1 = mk_tuple(atomy)
  statenew= _update(stateinitial,tagdelay,tmp1)
  tmp2 = _deref(statenew,tagdelay)
  tmpy' = tmp2("y")
  tmp3' = tmpy'*2
  statenewest = _update(statenew,tagdelay,tmp3')
  tmp5 = atomy + tmp3'
  tmp6 = _deref(statenewest,tagdelay)
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

Later transformation steps might create new dead code and variable renamings; these two steps have to be repeated a few times. Since they are cheap to run (mostly linear), this does not impose any costs.

8.5.3 Reference Tracing

A special-purpose optimization can be used to trace `_update`/`_deref` calls: a `_deref` can be removed (i.e., replaced by an identity for the updated atom) if the matching[36] previous `_update` is visible.

An `_update` can be removed (i.e., replaced by an identity for the state) if there is no following `_deref`, and a following `_update` on the same reference tag is visible.

These analyses will remove the first `_update` and both `_deref`s from the example code, which then is:

```
[ tmp1 = mk_tuple(atomy)
  statenew= stateinitial
  tmp2 = tmp1
  tmpy' = tmp2("y")
  tmp3' = tmpy'*2
  statenewest = _update(statenew,tagdelay,tmp3')
  tmp5 = atomy + tmp3'
  tmp6 = tmp3'
  tmp7 = tmp5 + tmp6
| tmp7 ]
```

Many new variable identities are introduced during this step, propagation and dead-code elimination leads to

```
[ tmp1 = mk_tuple(atomy)
  tmpy' = tmp1("y")
  tmp3' = tmpy'*2
  statenewest = _update(stateinitial,tagdelay,tmp3')
  tmp5 = atomy + tmp3'
  tmp7 = tmp5 + tmp3'
| tmp7 ]
```

8.5.4 Tuple Tracing

In analogy to reference tracing, tuple lookups can be removed if the matching `mk_tuple` creation is visible. Since most tuples are generated by lambda applications, tuple tracing is really an analysis of static variable binding times. This process removes references to the tuple and may lead to them becoming dead. Removing tuple lookups from the example changes the definition

[36] "Matching" means "referring to the same reference".

```
tmp_y' = tmp_1("y")
```

into

```
tmp_y' = atom_y
```

This introduces another variable renaming; it also removes the last use of $\texttt{tmp}_1$. Propagation and dead-code removal leads to

```
[ tmp_3' = atom_y*2
  state_newest = _update(state_initial,tag_delay,tmp_3')
  tmp_5 = atom_y + tmp_3'
  tmp_7 = tmp_5 + tmp_3'
| tmp_7 ]
```

A formal framework for reference- and tuple-tracing that goes beyond mere local scope optimizations can be found in the *path semantics* of Bloss [20], which is employed in [18, 19, 21] and [109] to gain usage information allowing safe destructive updates for arrays (or other structures).

This concludes the optimizations that are implemented in `ac` and apply to the example. To free the example from its last unnecessary `_update`, a non-local analysis is needed, since the reference $\texttt{tag}_{\text{delay}}$ might be used somewhere else in the program. This is impossible in the example (since `x`, to which the reference was bound, is not bound to a suspension and was never exported from its defining scope), but that information is lost at this stage of reconstruction. If `x` were bound to a suspension instead of a promise, removing its definition would change the semantics, since suspensions might wake up even when there are no references to them; this is not true for promises.

8.6 Example: Blocking Code

When a strict primitive encounters a reference argument that points to an unevaluated suspension, the computation *blocks*: a `Block` is created and a reference to it is returned.[37] The handling of `Block`s is one of the more complex tasks of the reconstruction phase, since it involves knowledge about the internal structure of the semantic functions. Since the treatment of `Block`s is so different from the treatment of the other suspensions (`Ususp`s, `Wsusp`s, etc.), an extra example is needed to demonstrate the generation of code for `Block`s. First however, an in-depth discussion of `Block`s is in order.

8.6.1 What's in a `Block`

A `Block` is a reference object similar to a pending suspensios (i.e., a `Ususp/Wsusp`). `Block`s are simpler than an ordinary suspension in that their condition is not an arbitrary closure; instead, it is fixed to `isAvailable(`*ref*`)` – a `Block` expresses a directly data-dependency; some expression has blocked because it needed access to a pending suspension. `Block`s also introduce extra complications since they allows no introspection: while ordinary suspensions contain `Lambda` closures, `Block`s contain "semantic continuations", i.e. a closure of an *interpreter* function.

The LF semantics is denotational by definition, but not in its spirit: the idea behind a denotational semantics (DS) is the association of language-level concept with their "intuitive" *mathematical* counterparts. For example, numeric objects should be modelled by (real or rational) numbers , and functions by (continuous)

[37] Some other expression contexts also enforce strictness: the condition of a `Lcond` or `Lselect` is strict, as are the types in function argument lists and `Lcheck/Lcast` nodes. Also, function arguments are strict if they are needed to resolve overloading. It might have simplified the semantics if the transformation phase had introduced a `strict_identity` primitive into the `Lcond` and `Lselect` nodes. Such an approach could, however, not be used to cope with the selective strictness needed for function arguments and parameter types.

functions. This is a fine approach to language modelling, since the whole power of standard mathematical notation can be employed in describing language features, but it creates a problem when efficient code has to be generated. In effect, a code generator for the mathematical theory that underlies the denotational semantics has to be written! Denotational semantics is, for all practical purposes, a typed lazy functional programming language. Much of the power of DS stems from the data types it provides, first amongst them (infinite-precision) numbers, (possibly infinite) sets and functions that need not be defined constructively. It may be harder to create a correct DS-based code generator than it would be to directly translate the language which is modelled by the DS.[38] While numbers and other "transparent" objects do not cause great problems, functions cannot be directly mapped to most machines. Amongst other things, functions are "opaque" semantic object type – if a function's domain is not finite, there exists no guaranteed way to describe a function using finite methods – it is not even possible to compare them for equality! While programming language functions are build from discrete components, mathematical functions can be defined in any number of ways.

The LF semantics tries to avoid opaque objects whenever possible. Functions are therefore modelled as code/environment tuples, in which the "active" part ($\texttt{apply}_{\text{closure}}$) is implicit and the same for each function. Suspensions are not modelled as pending continuations of the semantics, but as user functions. The only exception to this are the `Block`s. Sections 8.6.1.1 and 8.6.1.2 show two alternative strategies of implementing *Block*s that avoid semantic continuations, and give the reasons why they were not chosen.[39]

What exactly is a `Block`? First and foremost, the encapsulation of a semantic function continuation. There are about 50 occurrences of λ in the semantics; most (44) of them are motivated by the possibility of a `Block`, i.e. these are the possible "entry points" for a `Block`. Just like LF closures, semantic closures contain lexical variable binding. The "variables" are the parameters given to semantic functions, and the intermediate definitions used within the functions. A typical example for such a value is the partially-dereferenced list that is bound in the continuation of $\texttt{deref}_{\text{objs}}$.

8.6.1.1 Alternatives to Blocks: CPS

The first alternative to the current `Block` implementation is the adoption of continuation-passing-style (CPS). This approach also solves all problems related to exception raising. A CPS semantics was considered, but is *not* used, because code generation for such a semantics does face an entirely new problem: In a CPS semantics, there exist one explicit continuation function for (nearly) every non-atomic expression. During execution, this leads to the creation of zillions of closures, one for each continuation function. Indeed, CPS closures represent "first-class stack frames" of non-CPS implementations. The difference to ordinary ("second class") stack frames is that the latter have to be allocated and de-allocated in stacked order, while closures can be kept around for an arbitrary amount of time. Therefore, closures must be allocated on the heap, instead of on a stack.[40]

Mainly for this reason SML/NJ, which is based upon a CPS execution model, allocates about 1 word (4 bytes) every 3-7 instructions executed ([4], pp.196). SML/NJ achieves acceptable performance only due to its aggressive garbage collection scheme, which requires an amount of memory at least 3 times the size of the active set. When not simulating on a workstation, ALDiSP programs have to run on specialized DSP hardware in real-time; such an environment cannot support garbage collection. It might be possible to

[38] Of course, only one code generator has to be written for the DS, so the time needed to write a code generator for the target language has to be compared to the time needed to set up the DS of the target language. Literature describes some compiler-compilers that directly try to generate code from the DS; most of these can only generate adequate code when provided with a DS that employs tool-specific conventions (cf. [78]).

[39] Besides the problems related to code generation for *Block* creation and resumption that will be discussed now, there are consequences of the current *Block* implementation for the scheduler (these are described in section 7.7). For these reasons, blocking computations are to be considered the biggest mis-feature of the language.

[40] Many CPS run-time systems store closures on the stack and only export them to the heap when they escape their lexical scope. Such an optimization is suitable whenever most closures behave like ordinary stack frames.

eliminate the need for run-time creation of most closures by compile-time allocation/reclamation schemes (this is already done in `ac`), but I am not sure whether all closures introduced by a CPS-semantics can thus be eliminated.

8.6.1.2 Alternatives to Blocks: Syntax Transformations

The second approach to avoid `Block`s consists of a global syntactic transformation: any possibly blocking expression is transformed, e.g. any conditional

```
if x then y else z end
```

would be transformed into

```
let tmp_cond = x
in
 if isAvailable(tmp_cond) then
     if tmp_cond then y else z end
 else
     suspend
         if tmp_cond then y else z end
     until isAvailable(tmp_cond)
     within 0.0 ms, 0.0 ms
end
```

or, to reduce the code size if `y` or `z` are huge expressions,

```
let tmp_cond = x
    tmp_expr =_delay(lambda(){if tmp_cond then y else z end})
in
 if isAvailable(tmp_cond) then _deref(tmp_expr)
 else
     suspend _deref(tmp_expr)
     until isAvailable(tmp_cond)
     within 0.0 ms, 0.0 ms
end
```

There is also the possibility of a “direct” transformation without an extra `isAvailable` test:

```
let tmp_cond = x
in
  suspend
     if tmp_cond then y else z end
  until isAvailable(tmp_cond)
  within 0.0 ms, 0.0 ms
end
```

This is however incorrect in that it introduces extra indeterminism; the expression

```
let x = if true then 10 else 20
in
    if isAvailable(x) then 1
                      else 2
end
```

evaluates deterministically to `1` under the standard interpretation, but could deliver `1` or `2` after the transformation into:[41]

[41] We assume that a translator would not be so brain-dead as to ask `isAvailable(true)`

```
let tmpcond = true
    x = suspend
          if tmpcond then 10 else 20
        until isAvailable(tmpcond)
        within 0 ms, 0 ms
in
    if isAvailable(x) then 1
                      else 2
end
```

The correct transformations, if performed naively, would blow up the size of every program by a factor of 10 or more; if done "intelligently", e.g. by avoiding repeated availability tests for the same variable, this could be reduced to a mere 10-20%. However, it still smells of syntactic overkill.

There are also some cases that cannot be transformed correctly by syntactic means in a preprocessing phase, namely those that are data-dependent. To pick just one example, the types of arguments to a function must be known whenever the function is overloaded, since overload resolution works by testing argument types. Hence, some arguments of some functions can block applications. When functions are passed around as arguments, it is not possible to statically determine whether a variable is bound to an overloaded function or not.

8.6.2 The Blocking Example

Since the interior of a `Block` cannot be accessed, it is not possible to directly synthesize a reasonable code attribute for it. Instead, some "dummy code" is generated and refined in a post-processing step. The following example shall demonstrate this. This example is not explained in the detail of the first one; the exposition concentrates on the "problem areas".

Consider the ALDiSP program fragment

```
let
 x = suspend y+1 until true within 10.0 ms, 10.0 ms
in
 (x+42)+y
end
```

which translates to the LF code

```
let
 x = _suspend(Lambda().{y+1},
              Lambda().{true},
              ms(10.0),ms(10.0))
in
 (x+42)-y
end
```

the first part up to the `x+42` will translate into the declaration fragments[42]:

```
[ tupleexpr = mk_tuple(atomy)
  tuplecond = mk_tuple()
  tmpsusp   = _suspend(tupleexpr,tuplecond,10ms,10ms)
  statenew   = _update(stateold,refsusp,tmpsusp)
  ... ]
```

After evaluating the literal `+`, the first reference to `x` and the literal `42`, $\texttt{eval}_{expr/Lapp}$ calls $\texttt{apply}_{primitives}$ to apply the `+`. $\texttt{apply}_{primitives}$ invokes `deref_objs` to reduce its arguments to normal form, since `+` is strict.

[42] As the first example has shown, these declarations are distributed over many pending instantiations of $\texttt{eval}_{exprs}$ and $\texttt{deref}_{objs}$. For didactic purposes, we consider them as a whole set of already generated code pieces.

Now the reference $\texttt{ref}_{susp}$ (the result of evaluating `x`) is encountered. $\texttt{deref}_{objs}$ blocks; the return value is a new reference ($\texttt{ref}_{block}$) bound to a $\texttt{Block(ref}_{susp}\texttt{,f}_{magic}\texttt{)}$ suspension. The function $\texttt{f}_{magic}$ encapsulated in the `Block` is a continuation of the semantic function apparatus itself, and cannot be further analyzed. It can, however, be modelled as if it were an ordinary, if unknown, `Lambda` expression!

When, later on, the blocked evaluation thread continues, this takes place in an evaluation context similar to the current one (basically, it is the same context, but with the then-current state). Any code that will be generated for the continuation will therefore need the current context. Code that models the preservation of the current context must therefore be created. In analogy to the standard closure creation, this is modelled by a `mk_tuple` application that binds the current lexical context. In analogy to `_suspend`, a `_block` primitive models the creation of a "block object" that is stored in the state:

```
[ tuple_expr = mk_tuple(atom_y)
  tuple_cond = mk_tuple()
  tmp_susp   = _suspend(tuple_expr,tuple_cond,10_ms,10_ms)
  state_new   = _update(state_old,ref_susp,tmp_susp)
  tup_blk1   = mk_tuple()
  tmp_blk1   = _block(ref_susp,tup_blk1)
  state_newer= _update(state_new,ref_blk1,tmp_blk1)
| ref_blk1 ]
```

The problem is to determine the set of variables that have to be captured in the `mk_tuple`. At the point where it is generated (i.e., in the semantic functions `block` and $\texttt{deref}_{objs}$), no information is available about the variables captured in the `Block`. The only "upper limit" is set by the fact that the captured variables must come from the current Code Form context. The set of lexical variables is determined syntactically by collecting all free variables of the `Code` expression up to the point of the `_block`. This list of bound variables is not readily accessible at the point where the `Block` is created; therefore a hack is employed: the `mk_tuple()` application is initially left empty; whenever a `Code` object is "passed upward" and recombined with other `Code` objects, any `mk_tuple` that serves a `_block` is patched with the CF variables that become visible through the recombination.

Evaluation continues with the evaluation of `y`, followed by the application of `-`. Skipping details, $\texttt{deref}_{objs}$ will encounter the reference argument that points to $\texttt{tmp}_{blk1}$. This gives rise to a second `Block`, bound to a new reference ($\texttt{ref}_{blk2}$) and with code

```
[ tuple_expr = mk_tuple(atom_y)
  tuple_cond = mk_tuple()
  tmp_susp   = _suspend(tuple_expr,tuple_cond,10_ms,10_ms)
  state_new   = _update(state_old,ref_susp,tmp_susp)
  tup_blk1   = mk_tuple()
  tmp_blk1   = _block(ref_susp,tup_blk1)
  state_newer= _update(state_new,ref_blk1,tmp_blk1)
  tup_blk2   = mk_tuple()
  tmp_blk2   = _block(ref_blk1,tup_blk2)
  state_newest=_update(state_newer,ref_blk2,tmp_blk2)
| ref_blk2 ]
```

Evaluation control returns to $\texttt{eval}_{expr/let}$, which creates no code of its own and simply terminates. When the enclosing function, of which the example is a sub-expression, terminates, the two `mk_tuple` creations are updated to

```
[ ...
  tup_blk1    = mk_tuple(tuple_expr,tuple_cond,tmp_susp)
  tmp_blk1    = _block(ref_susp,tup_blk1)
  state_newer= _update(state_new,ref_blk1,tmp_blk1)
  tup_blk2    = mk_tuple(tuple_expr,tuple_cond,tmp_susp,tup_blk1,tmp_blk1)
  ...
| ... ]
```

When abstract time will have advanced by 10 milliseconds, the abstract scheduler decides to evaluate the `Ususp` bound to $\texttt{ref}_{susp}$. This happens at the state-function level, i.e. the evaluation and the following update make up a state transformation. It is guaranteed that all suspensions are only evaluated once, so the code generated for suspension evaluations and block continuations can be inlined, and makes up the entire state transition function.

$\texttt{eval}_{thunk}$ is called with the LF function λ`().y` and the arguments $\texttt{tuple}_{expr}$ and $\texttt{state}_{later}$ (the then-current state). The $\texttt{tuple}_{expr}$ was found by looking up the $\texttt{ref}_{susp}$ in the state. The optimized (inlined etc.) result has as code

```
[ tup_susp = _lookup(state_later,ref_susp)
  tmp_y = tup_susp("y")
| tmp_y ]
```

The `atom` that was associated with the abstract object bound to `y` is discarded.[43] An `_update` is generated for the $\texttt{ref}_{susp}$; due to the inlining it can be directly appended to the thunk code:

```
[ tup_susp = _lookup(state_later,ref_susp)
  tmp_y = tup_susp("y")
  state_later' = _update(state_later,ref_susp,tmp_y)
| state_later' ]
```

The result of the state transforming function is specified is the new state.

The abstract scheduler can now evaluate the first `Block`. It does so by creating a preamble that sets up the variables caught in the $\texttt{tuple}_{block1}$, and calling the semantic continuation with the current state ($\texttt{state}_{later}$'). The continuation incorporates this state into the context that was active at the blocking point, and continues evaluation of the blocked semantic function, in this case $\texttt{deref}_{objs}$ of the $\texttt{ref}_{susp}$, as if nothing had happened. This reference is now bound to the abstract object of the original `y`; the application of `+` can continue. The continuation function terminates with the return from $\texttt{eval}_{expr/Lapp}$, and with the code

```
[ tmp_x = _lookup(state_later',ref_susp)
  tmp_r = tmp_x + 42
| tmp_r ]
```

After the preamble and the update to $\texttt{ref}_{block1}$ are added to this, the code of the `Block` is

```
[ tup_block = _lookup(state_later',ref_blk1)
  ...
  name_var = tup_block("name_var") --- for all variables catched in tup_block
  ...
  tuple_expr= tup_block("tuple_expr")
  tuple_cond= tup_block("tuple_cond")
  tmp_susp   = tup_block("tmp_susp")
  tmp_x = _lookup(state_later',ref_susp)
  tmp_r = tmp_x + 42
  state_later'' = _update(state_later',ref_blk1,tmp_r)
| state_later'' ]
```

[43] If the definition of said atom is visible in the scope where the result of the tuple is used, reference- and tuple-tracing will find it.

This is the complete code to be included in the state transformation function for the first block.

Evaluation of the second `Block` proceeds in the same manner, resulting in code

```
[ tup_block2 = _lookup(state_later'',ref_blk2)
  ...
  name_var = tup_block2("name_var") --- for all variables catched in tup_block2
  ...
  tuple_expr= tup_block2("tuple_expr")
  tuple_cond= tup_block2("tuple_cond")
  tmp_susp  = tup_block2("tmp_susp")
  tup_blk1  = tup_block2("tup_blk1")
  tmp_blk1  = tup_block2("tmp_blk1")
  tmp_res1  = _lookup(state_later'',ref_blk1)
  tmp_r = tmp_x + y
  state_latest = _update(state_later'',ref_blk2,tmp_r)
| state_latest ]
```

The `code` attributes that are generated for the `Block` continuations are quite inefficient due to the unnecessary `tuple(...)` applications. It is especially important to notice that many blocks will be directly dependent upon other blocks, and thus be evaluated in sequence. When the two state-transition functions that model the blocks' evaluation are inlined[44], dead-code- and parameter-elimination will take care of most of these inefficiencies.

An important optimization that simplifies the generation of code both for the `Block` tuple creation, and for the tuple "unpacking" that takes places as preamble to the continuations' code sequences, is the *lazy variable capture*. After a `Block`'s transition function has been created, the variables that were needed from the lexical environment of the `Block` creating site can be deduced from the code, since these will be those variables that are referenced, but undefined. It is possible to "patch" the original `mk_tuple` and the preamble at this point, since now it is now longer necessary to put a conservative approximation of the current context into the `Block` tuple, but only those variables actually needed. This optimization can greatly decrease the intermediate code size.

8.7 Specification of Code Generation

To specify the code generation semantics of the compiler, it is necessary to extend the definitions of the semantic functions ($\mathtt{eval}_{expr}$, $\mathtt{eval}_{decl}$, `apply` etc). This effort can be minimized by employing the fact that only a few clauses of the semantic functions are responsible for generating new abstract results, and hence new code. Also, whenever the semantics employs a literal result (e.g., "true" or "false"), the `code` attribute is trivially defined by an `ALit()` atomic expression.

All functions that isolate *Obj*s from *Result*s and pass on the *Obj*s to continuation functions must re-attach the original *Result*s `code`'s declaration list to the `code` of the *Result* that is returned by the continuation function.

The only "complex" code generation is related to the application of closures and primitives.

All other clauses (i.e., those that just inspect *Result*s and pass them through unchanged) can ignore the `code` attribute. In the following, extensions for the semantic functions $\mathtt{deref}_{obj}$, $\mathtt{eval}_{expr}$, `apply`, and $\mathtt{eval}_{decl}$ are given. Most extensions are specified informally.

[44] Since the first transition function pass its results to a state function that directly calls the second transition function, inlining will merge direct sequences of `Block`s.

8.7.1 $\texttt{eval}_{\text{exprs}}$

This auxiliary function repeatedly calls $\texttt{eval}_{\text{exprs}}$, strips the objects from the results, and passes the object list to a continuation. Since it has also stripped the `code` attribute from the results, $\texttt{eval}_{\text{exprs}}$ is responsible for appending the results' CF declaration lists to the code of the result that is returned from the continuation. If no lazy `Block` code generation is used, $\texttt{eval}_{\text{exprs}}$ is also responsible for patching any `mk_tuple` applications that are used to create blocks.

8.7.2 $\texttt{deref}_{\text{objs}}$

The $\texttt{deref}_{\text{objs}}$ function directly analyzes the types of objects. If an object is a reference, the referenced value is substituted. The code fragment for this case is simply a `_deref` primitive application. If the referenced value is a unevaluated promise, the thunk has to be evaluated and the result value to be stored:

```
[ tmp_thunk  = _deref(state_current,ref)
  tmp_result= tmp_combinator(tmp_thunk)
  state_new  = _update(state_current,ref,tmp_result)
| tmp_result ]
```

The $\texttt{tmp}_{\text{thunk}}$ is represent the closure tuple of the promise. It has to be passed explicitly to the promise's combinator, as the first (and sole) argument. The "code", i.e. the combinator, is treated like a literal or a top-level definition.

8.7.3 `block`

Most of the behaviour of the `block` function has already been described in the second example (section 8.6). Block dumps the initially empty tuple creation code and the definition of the reference that points to the block:

```
[ tmp_tuple  = mk_tuple()
  tmp_block  = _block(ref_pending,tmp_tuple)
  state_new  = _update(state_old,ref_block,tmp_block)
| ref_block ]
```

The reference $\texttt{ref}_{\text{pending}}$ is the one that caused the block $\texttt{ref}_{\text{block}}$ is the one that denotes the block.

8.7.4 `strict` *and* $\texttt{strict}_{\text{exprs}}$

$\texttt{strict}_{\text{exprs}}$ just combine $\texttt{eval}_{\text{exprs}}$ and $\texttt{deref}_{\text{objs}}$. `strict` extracts an object from a result and passes it on to a continuation function; it therefore has the responsibility to re-attach the stripped CF declarations to the result that is passed back.

8.7.5 $\texttt{eval}_{\text{thunk}}$ *and* $\texttt{norm}_{\text{result}}$

$\texttt{eval}_{\text{thunk}}$ combines $\texttt{norm}_{\text{result}}$ and $\texttt{deref}_{\text{objs}}$. As already been mentioned in section 8.3, $\texttt{norm}_{\text{result}}$ has been modified to return a true *Result*-typed result; therefore no code generation takes place.

8.7.6 *Lit*

The `code` for a `Lit` expression is an emplty declaration sequence with an `ALit` return expression:

```
[ | ALit(value) ]
```

8.7.7 Lexical Lvars

A lexical `Lvar` expression has as its code the `atom` that was bound to the value of the lexical variable. This atom has been set up when the variable was defined.

There are three different "places of origin" of lexical LF variables:

- `Lambda` parameters,
- variables bound with `Ldecl/Lpard/Lseqd/Lfixd`, and
- variables from an enclosing scope, which are extracted from the closure tuple.

`Lambda` parameters and closure tuple variables are routineley given fresh names in each function's preamble, if only to simplify inlining. LF declarations are effectively ignored at the CF level, since they are kept track of by the compile-time environment.

8.7.8 Dynamic Lvars

In general, dynamic variables are either passed to the function (that encloses the `Lvar` expression) as an additional parameter (hidden in the state variable). The generated code employs a special primitive `_lookup_dynamic` that accesses the current state:

[$\texttt{var}_{\mathrm{dyn}}$ = `_lookup_dynamic(`$\texttt{state}_{\mathrm{current}}$`,"`$\texttt{name}_{\mathrm{dyn}}$`")`
| $\texttt{var}_{\mathrm{dyn}}$]

When a dynamic variable is bound to a literal value, i.e. either a "true" literal or a combinator function, *function specialization* can take place. The dynamic variable essentially disappears, at the possible cost of an additional function specialization.

8.7.9 Lambda

The evaluation of a `Lambda` expression creates a closure object; its run-time equivalent is a closure tuple. This tuple is created using a `mk_tuple` application that binds all currently visible variables that will be used in the tuple's body:

[`tup = mk_tuple(`$\texttt{atom}_{\mathrm{free}/1}$`,....,`$\texttt{atom}_{\mathrm{free}/n}$`)`
| `tup`]

Each $atom_{free/i}$ is the `atom` attached to $var_{free/i}$, the ith free variable in the `Lambda` expression.

The set of free variables in the body of the `Lambda` expression that can be determined statically is a conservative approximation: it may contain variables that are never used by the actual execution.[45] One useful optimization of the code generator is therefore the deferral of actual tuple creation to the point where all applications of the tuple are known; then the worst-case variable set can be deteremined and used to patch up the tuple creation code.

8.7.10 Lcheck

`Lcheck` forms reduce to a either a function application, or to the creation of a new `Lambda` expression, i.e. a tuple (if a function-type is checked for). If the `Lcheck` reduces to a function application that generates a non-abstract value, no code is generated at all: if the result is `true`, the `Lcheck` is replaced by its argument expression.; if the result is `false`, a compile-time error is issued. If the result is an abstract value (i.e., `aBool`), a runtime test may be generated. The current `ac` does not generate run-time tests; this is correct, since the outcome of an unsuccessfull test is undefined.

[45] Actually, if the closure is never applied at all, no variable is used or need be captured.

8.7.11 *Lcast*

Evaluation of an `Lcast` will lead to an application of one of the user-defined cast functions. If a cast to a general type is needed, and no unique matching type can be found, because there are type predicates that return `aBool`, nested run-time conditionals have to be generated: one for each possibly applicable cast function that precedes the first guaranteed applicable cast function.

8.7.12 *Lcond*

A conditional that evaluates to `true` or `false` will be replaced by the chosen alternative. If the condition is undetermined, both alternatives are evaluated, and a `Select` node with two forks is created from the code of the condition and the alternatives:

```
[ ...code that computes the atom_cond ...
 (result,state') = Select(atom_cond,
                          [(ALit(true), [... decls_true ... | result_true,state_true]),
                           (ALit(false),[... decls_false ... | result_false,state_false])],
                          NONE)
 | result ]
```

8.7.13 *Lselect*

The only `Lselect` nodes that appear in practical programs are generated in module selection functions. In this context, it is an error if no compile-time selection can be determined. Therefore, no `Select` code must be generated, since one of the selection cases, or the default, will be chosen.

8.7.14 *Lseq*

The evaluation of a sequence is for its side-effects only. The *strict* function implements the evaluation of (and therefore also the code generation for) the expression list; the `eval(Lseq(...))` merely discards all but the last result; the nested *strict* applications will prefix all the code generated for the sequence. Any unnecessary code will have to be removed by post-processing optimizations.

8.7.15 `apply`

Most of the sub-functions of `apply` do not generate code, they just decide which of the many $\texttt{apply}_{\text{xxx}}$ function is to be used. At the bottom, there are three different functions that implement the application:

- $\texttt{apply}_{\text{type}}$, with a type argument that is one of the basic types,
- $\texttt{apply}_{\text{primitive}}$, and
- $\texttt{apply}_{\text{closure}}$.

Only one of the "distribution" functions can generate code, namely $\texttt{apply}_{\text{overloaded}}$. While the actual application of $\texttt{apply}_{\text{overloaded}}$ reduces to an $\texttt{apply}_{\text{closure}}$ (since the overloaded function is just a collection of ordinary closures), the selection of the closure might be deferred to run-time. In this case, a nested conditional is created. In the semantics, the $\texttt{check}_{\text{list}}$ function is used to determine the index of the first matching closure

of an overloaded function. This function has to be modified upon abstraction, since it can only return `true` or `false` in its original form.[46]

Some code has to be generated for overloaded function applications, namely the selection of the closure that is to be applied. The runtime model of an overloaded function is a tupel that contains closures. The primitive `overload` models the creations of such a tuple, and `overload_select` models the closure selection.

8.7.16 $\texttt{apply}_{\text{type}}$

Code generation for $\texttt{apply}_{\text{type}}$ (or rather, $\texttt{is}_{\text{base}}$) is trivial, since the results must be either `true` or `false`, if the type that is tested for is a base type. The runtime system does not know data polymorphism, i.e. every variable in the compiled CF program is of a known and fixed base type. All polymorphic values are split up as a side effect of function specialization.

When $\texttt{apply}_{\text{type}}$ is used to test for a predicate type, it reduces to a closure application.

8.7.17 $\texttt{apply}_{\text{primitive}}$

Primitive function applications fall into two categories:

- the "simple" primitives are strict and side-effect free; examples of these primtives are the arithmetic primitives, and the tuple accessor and creator primitives. Code generation for these primitives is trivial: $\texttt{apply}_{\text{primitive/strict}}$ looks at the result value; if it can be encoded as a literal, the result `code` is this literal; if it is a non-literal, the code is

  [result = Prim(primitive-name,[$\texttt{atom}_{\text{arg/1}}$,...,$\texttt{atom}_{\text{arg/N}}$])
  | result]
- the "complicated" primitives are non-strict and/or have side effects. In general, code generation for these works as in the "simple" case, but some of the side-effecting primitives have special-case code generation rules:
  - `mk_exc` is translated into a `Throw` expression. `mk_exc` takes an arbitrary number of argument; the corresponding `Throw` has one additional argument atom, namely the current state (which is the first argument). The result of the `Throw` is a new state.[47]
  - `suspend` and `delay` primitives are translated into sequences of a `_suspend`/`_delay` that creates a suspension/promise object, and an `_update` that installs the newly created object in the current state, and creates a new state. (This has been shown in the examples.)

[46] In the current implementation, code generation for the generalized $\texttt{apply}_{\text{overloaded}}$ is not implemented, and an error message is issued when an overloaded function application cannot be resolved at compile time. In practice, most of these reflect programming errors anyhow.

[47] This is really a pseudo-result, since the `Throw` does not return, but jumps to the last "Catch" on the call stack. There is an additional problem with `Throw`, which is exemplified by the common idiom

```
[ t = eq(b,0)
  S'',q = if t then [ S' = Throw(S,DivZero,a) | S' ]
               else [ r  = div(a,b) | S,r ]
| q ]
```

There is an obvious mismatch here. Semantically, `mk_exc` and `Throw` return exception objects, which can be treated like any other bottom-valued object. At the CF level, there is no `bottom`, and variables are restricted to known (base) types. To guarantee a type- and arity-correct CF program, each `Throw` therefore returns an extra dummy-value that is typed accordingly to the needs of the context. These needs become obvious the first time the object is merged with other (non-bottom) objects in a conditional. The correct code is therefore

```
[ t = eq(b,0)
  S'',q = if t then [ S',dummy = Throw(S,DivZero,a) | S',dummy ]
               else [ r  = div(a,b) | S,r ]
| q ]
```

- the `map_array` and `map_list` primitives are translated into library function calls.
- the `io_op` primitive is translated into an `_update`. `io_op` is a general I/O operation that can be parameterized to effect input, output, and test operations. In any of these cases, reference definitions have to be created or updated.

8.7.18 $\texttt{apply}_{\text{closure}}$

When a closure is applied, a lot of code has to be created. Each closure application is assumed to create a new call cache entry (CCE). Each CCE will either be inlined, or projected to a CF function. Code generation assumes the latter.

Code generation consists of two tasks: the `atom` attributes of the values held in the (updated) environment have to be modified, and a preamble (that implements these modifications) has to be prepended to the final `code` of the function's result. To pick an example function

```
c = ...
func f(a,b) = a+b*c
...
f(x,y)
```

The closure `f` contains one value, namely `c`. Upon application of `f`, code for three bindings (`x`,`y`, and `c`) must be created. The application takes the form

$$\texttt{r = App(tag}_{\text{cce}}\texttt{,atom}_{\text{f}}\texttt{,atom}_{\text{x}}\texttt{,atom}_{\text{y}}\texttt{)}$$

where $\texttt{tag}_{\text{cce}}$ is the tag that denotes the call cache entry in which the code for this particular application of `f` will be stored, and $\texttt{atom}_{\text{f}}$, $\texttt{atom}_{\text{x}}$ and $\texttt{atom}_{\text{y}}$ are the `atom` attributes of the variables `f`, `x` and `y`.

For the CCE code, four new variables are introduced: $\texttt{tmp}_{\text{tuple}}$, $\texttt{tmp}_{\text{a}}$, $\texttt{tmp}_{\text{b}}$, and $\texttt{tmp}_{\text{c}}$. Of these, all but $\texttt{tmp}_{\text{c}}$ are parameters to the function. $\texttt{tmp}_{\text{tuple}}$ is the parameter to which `f`'s closure tuple is bound, and $\texttt{tmp}_{\text{a}}$/$\texttt{tmp}_{\text{b}}$ are the two "ordinary" parameters.[48] The preamble is such quite small, namely

```
[ tmp_c = lookup_tuple(tmp_tuple,1)
  ... rest of the code ...
| ... results ... ]
```

The names of the newly introduced parameters are stored as attributes of the CCE, since they will be needed when the CCE is later inlined or translated into a function definition.

Again, no code has to be generated at all if the result is a literal value. In such a case, the call cache entry is reclaimed.

8.8 Related Work

This section gives a short comparison with the approaches to residual code generation and representation that are presented in the literature. Since most published work describes off-line PE systems, these shall be presented first.

Off-line partial evaluation is usually specified as a source-to-source transformation that is guided by annotations generated in a binding-time analysis (BTA) phase. These annotations typically take the form of a two-level syntax.[49] The essential fold-unfold transformations [22, 23], together with typical λ-calculus identities like α-conversion, form a sufficient base for all modifications encountered in off-line PE; i.e. all "higher

[48] These parameters do *not* have the names `a` and `b`, since inlining would be complicated by such a renaming. Besides, debugging is kept simpler if there are no common names between the LF and CF domains.

[49] In a two-level syntax, there are two variants for each "original" syntactic construct. In BTA, these variants denote the "static/dynamic" distinction (cf. section 5.6).

level" optimizations can be specified and implemented in terms of these basic transformations.[50] Most of the complexity of off-line PE is isolated in the BTA, and program representation doesn't become an issue at all. This is indeed one of the strengths of the off-line approach: the separation of BTA and rewriting makes it possible to simplify both phases; BTA can be implemented using simple-minded, but still useful data models (e.g., AI on two-point domains), and the rewriting phase needs nothing more then $\alpha-$ and $\beta-$conversion (renaming and instantiation).

In on-line systems, the program is re-written while it is executed. An on-line PE must therefore consist of at least a full interpreter; usually even an "abstracted" version of the standard interpreter.[51]

Code generation in on-line PE also poses a challenge to software engineering because of is *distributed* nature. While the object program is executed, either code *fragments* are generated, or the object program is locally rewritten. In both cases, there is not much information available about the "whole" program, especially about the execution of things that are still in the "future". This is a marked contrast to off-line PE, where it is known by syntactical means whether a variable is used once or many times, passed upward or downward, or stored in a data-structure. For each of these situations, a different code generation approach can be employed. In on-line PE, nothing is known about the future of locally computed values, and the code generation strategy must therefore assume a worst-case context.

There are two essential approaches to code representation that can be characterized as being based on program *texts* versus program *graphs*. A "text representation" shares all the properties of a program in the source language, or its abstract syntax tree equivalent; it obeys the normal scoping and life-time rules, and it is implicitly sequentialized. Graph-based program representations tend to resemble data-flow graphs and are usually taken to be demand-driven; some graph representations allow explicit cycles, others forbid them or allow only directed acyclic graphs (DAGs). Graph representations have no natural scoping and sequencing rules; that makes them particularly suited for parallel applications [120].

Berlin [13, 14] employs a DAG representation for his LISP-based PE. The graph is then scheduled for optimized utilization on pipelined architectures. Berlin considers typical "scientific" applications that have a fixed recursion structure dominated by numeric computations; he essentially optimizes inner loops.

The FUSE system of Weise et.al. [103, 121, 120, 122] employs a graph representation that allows cycles. FUSE is partly off-line and partly on-line. During abstract interpretation, a data/program graph is created; each data object is attributed with the code that generates it. The graph structure emerges as a natural consequence of these attributes. The reduce/residualize decisions are delayed to the partial evaluation phase that occurs after the abstract interpretation. In this phase, those applications that fulfill certain heuristically determined qualifications are reduced. This phase is simplified by the graph representation, since scoping can be ignored. Since source and target language are the same in FUSE, scoping has to be re-created, and the operations in the program graph have to be sequentialized. This is done by computing a def-point tree for the whole graph, from which scope information for each node is inferred. This scoped graph is then translated into a text representation (a Scheme program). Alternatively, the scoped graph can be translated into C. These tasks are not simplified by the possible need for nested and local higher-order function definitions.

In [123], Weise et.al. apply a similar representation to C programs. Loops and labels are translated into `Lambda` entry points. Since C does not provide nested first-order functions, the scoping structure is easier to reconstruct than in FUSE. The main goal in this work is to extract parallelism by removing implicit sequencing. To do this, the store is split up and some pointer analysis is performed.

Haraldsson [56] was one of the first to build an on-line partial evaluator as an extension to an interpreter. In the REDFUN-2 system, he introduces "q-tuples", which "[hold] the reduced form and information extracted

[50] Off-line PE can thus be characterized as an approach which tries to solve the problem of *what* to transform, not *how*.

[51] On-line PE using a standard interpreter is nothing more or less than run-time code generation (RTCG)! RTCG is feasible for some specialized applications such as sound and image decompression or BITBLT operations (cf. Keppel et al. [68] for examples). It is however hard to justify run-time code generation as a general strategy for the compilation of whole programs, since the increase in runtime efficiency that is achieved by the PE must compensate for the memory and time consumption of the partial evaluator – costs that can be ignored when run-time and compile-time are different.

from it [...]". REDFUN-2 is implemented in the context and as an extension of an existing LISP system, and the "reduced form" is a simple LISP expression; the "information" consists of a side-effect flag, true- and false-context information, and an "assignment information". The reduced forms are to be placed in `PROGN` contexts; hence variable definitions will be visible to all subsequent reduced forms. Inlining is performed ad-hoc and locally, guided by heuristics. Since REDFUN is embedded in a LISP environment, the user can extend and change the behaviour of many parts of the partial evaluation functions by adding properties to symbols. Using such explicit modifications of the partial evaluator, particularly bad specialization- and inlining-decisions can be avoided on an ad-hoc basis, and heuristics can be adapted to specific target programs.

Chapter 9

The Back-End

This chapter describes syntax and semantics of the generalized machine language M that is the final output of the compiler. M is a low-level language designed to be easily translatable to the assembler languages of current DSP processors. The current implementation of `ac` does not generate code for a specific target architecture; instead, the user can request the generation of either an M code program, or a portable `C` program that corresponds to it.[1]

9.1 *Differences between* M *and CF*

Why is it necessary to have yet another intermediate form? The Code Form representation is still too "high level" in that it is based on the applicative paradigm. The control flow of CF programs is based on tail-recursive function calls instead of jumps, loops, and stack-based subroutine calls. All CF data structures, including tuples, arrays, and the state, are first-class values: they can be passed around as function arguments and result values. On a real machine, only "atomic" values small enough to fit into registers have this first-class status.

The M form is defined in terms of a machine model that allows easy translation to most currently known DSP and general purpose processor architectures. To achieve this goal, the abstract target machine is restricted in a number of a ways:

- No *flat address space* is assumed, i.e. there is no general pointer concept.
- A special *index type* is used to access vector elements.
- A special *loop construct* is provided.
- There is *no unique data size.*
- There is *no implicit control state.*

In the next sections, these points will be elucidated.

[1] It is also possible to generate a dump of the final call cache contents in Code Form, and a graphical rendering of the state graph in `GraphEd` format (for an example, see fig. 7.1).

9.1.1 Disjoint Address Space, Index Types, and Hardware Loops

Many DSP architectures, e.g. the DSP56k family of processors, have the capability of addressing two or more separate memory banks. This makes it possible to access more than one memory location in one cycle. Many common DSP operations (e.g, most filters) have a core of multiply/add loops, in which one word from the coefficient table and one word from the input stream are multiplied per iteration step, and the result added to an accumulator. Having two or more memory banks available in parallel makes it possible to implement one such MAC (multiply-accumulate) step per cycle. Languages such as "C" which depend upon a flat memory model with unrestricted pointers are not easily adapted to such architectures.

M supports target architectures with separate memory banks by *not* providing a general pointer type. As far as M is concerned, each vector (and each global scalar) can be part of a separate memory bank, accessed by distinct instruction addressing modes.

Since M shuns pointers, there are no alias-problems; all accesses to a given vector can be found by static analysis. Thus, it is simple to distribute the vectors over the memory banks.

A related problem is that of *index ranges*. M does not allow the indexing of arrays by ordinary unrestricted integers. Instead, it is possible to define a variable as being an "index into" one or more specific vectors. Such an index can only be used to access the vectors over which it is declared, and a vector can only be accessed by an appropriate index.

It is allowed to copy index values into "ordinary" integer variables of appropriate size, and vice versa; index values can also be incremented and decremented by integer amounts to give new index values. It is thus possible to directly map "index variables" onto target hardware index registers, if such exist.

Many DSP processors support "hardware loops", i.e. efficient increment/compare/jump combinations; sometimes there are even advanced features such as modulo-addressing. Such loop instructions are usually restricted to situations in which the number of loops and the first index are known in advance, and the step size is a small constant. To support such instructions, M contains a `loop` construct that is restricted in such a way that hardware loops for most DSPs can be generated without undue contortions.

9.1.2 Data Formats, Special Instructions, and the ALDiSP Library

The size of integer- and floating-point numbers is not standardized for all target architectures. Especially on DSPs, word sizes such as 12, 18, or 56 bit are not uncommon. No efficient code for such machines could be generated for an intermediate language that enforced one standard size such as 16 or 32 bit. On the other hand, if one "machine int" of unspecified length is used, the code would have no defined semantics, and machines supporting more than one integer or floating-point format could not be utilized. M therefore allows the specification of arbitrary integer and floating point formats.

The ALDiSP library is responsible for breaking the arithmetic primitives down to those sizes that are supported by the target architecture. This approach also unifies multiple-precision arithmetics with "ordinary" arithmetics.

A set of ALDiSP primitives defines the arithmetic capabilities of a machine.[2] This way, even user-written (library or program) functions can inquire about machine characteristics. Much of the complexity of a typical back-end and run-time support system is thus delegated to the library.

The ALDiSP library is required to only generate calls to primitives with arguments that are supported by the hardware; this guarantees that the generated M program is semantically sound.

If the target machine supports special instructions such as multiply-and-accumulate, or implements ALDiSP primitives as combinations of multiple machine instructions (e.g. multiplication as sequence of booth steps),

[2] Two primitives suffice: `_is_mach_type` asks whether a type is supported by a given target architecture, and `_nearest_mach_type` gives the target-type that is "nearest" a given type.

the ALDiSP library can be re-written to support those. The M language treats the primitive functions as "black box" pure functions. It is assumed that the back-end supports all instructions generated by the front-end, i.e. that the front-end knows what primitives are available on the target.

9.1.3 No Implicit State

Because M supports multiple output operations, there is no need for implicit state in form of status flags or the like. This simplifies M-level program transformation: instructions can be moved around as long as all data dependencies are maintained. M primitives are fully "functional" - they effect nothing beside their explicit output values. This is very important for instruction scheduling, which depends on the ability to freely move code around in the program.[3]

9.2 Semantic Entities of M

M assumes a machine with a main memory consisting of arbitrary many (but compile-time constant bound) disjoint vectors of scalar elements. A stack of known maximum size is supported. Program memory is separated from data memory. There is an unbound but fixed number of registers referred to implicitly. Basic scalar types supported are

`int(`n`)` for $n \leq \mathit{MaxIntSize}$, where $\mathit{MaxIntSize}$ is a machine specific constant. These are the twos-complement integers of size n.

`card(`n`)` for $n \leq \mathit{MaxCardSize}$ (which usually coincides with $\mathit{MaxIntSize}$). These are the unsigned integers of size n.

`float(`n, m`)` where (n, m) are drawn from a machine specific set of known float formats. Usually, the formats $\{(16, 8), (32, 8), (48, 16)\}$ are supported. An encoding in IEEE-754 floating point format is assumed.

`bool` is the standard boolean type.

`index(`*names*`)` is the type of array indices. An index is restricted to point into the arrays enumerated in its declaration. There are no pointers into arrays. Index variable may not point outside arrays. Not all indices need be of the same size, thus they are not assignment-compatible. An assignment $\mathtt{i}_1$ `=` $\mathtt{i}_2$ is allowed if the set of arrays $\mathtt{i}_2$ may point to is a subset of the set of arrays $\mathtt{i}_1$ may point to (that is, every valid $\mathtt{i}_2$ is also a valid $\mathtt{i}_1$).

There are the following basic kinds of instructions:

- *Declarations* introduce I/O objects, functions, and global variables. Within functions, they introduce stack variables and temporaries.
- *Assignments* move scalar values between locations. Assignments are instantaneous. Multiple assignments are simultaneous (`(a,b)=(b,a)` is a correct swap).
- *Primitives* can be called with any number of arguments, returning any number of arguments.
- *Jumps* can move between labels of the same function.

[3] The task of instruction scheduling is to find an ordering for the instructions that minimizes register spilling and optimizes the use of memory accesses and delay slots; this is quite important if the machine supports "parallel moves", as the 56k/96k processors do. The current ac considers instruction scheduling a machine-specific "back end" issue.

- *Calls* are used to enter another function.
- a *Return* exits from a *call*ed function, destroying all stack variables defined in that function.

Both jumps and calls allow argument and result parameter transfer (by value).

9.2.1 Function Definitions

A *function* is a piece of code containing at least one return statement. A function's definition ends with the header of the next function's definition. *The header argument list is empty.* Following the function header, and preceding the first executable statement or label, are declarations of local stack variables. Whenever the function is entered with `call`, a stack frame containing these variables is allocated, but not initialized. The `call` may or may not initialize some of these variables (namely, those mentioned in the parameter list).

A *temporary* variable is introduced by a `temp` declaration. The lifetime of the variable is up to its last use. It is illegal to use temporary variables in recursive code. Temporary variables are intended to model registers; the reconstruction phase tries to honor this intention by preferring temporary to stack variables.

A function can have arbitrary many *labels*. Labels have argument lists containing only the names of locally visible stack or temporary variables.

Control flow within a function is done via combinations of the `if` and `jump` directives.

As examples, here are two implementations of a well-known algorithm:

```
func fak1(n)          ;; recursive fak
  stack int(16) (n)   ;; n has to be on the stack
  temp int(32) res    ;; the result need not be on the stack
  temp bool (r)       ;; holds the comparison result flag
  temp int(16) n_1    ;; holds the n-1 value
  r = ">" (n,1)
  if (r)
    n_1 = "-" (n,1)
    res = call fak1(n_1)
    res = "*" (res,n)
    return (res)
  else
    return (1)
  endif
func fak2(n)          ;; iterative fak
  temp int(16) (n)    ;; input and loop counter
  temp int(32) (res)  ;; accumulator and result
  temp bool r         ;; for the comparison result flag
  res = 1
label loop()
  r = ">"(n,1)
  if (r)
    return(res)
  else
    res = "*"(res,n)
    n = "-"(n,1)
    jump loop()
  endif
```

It is important to understand why the "`n`" variable of the `fak1` function *must* be of type "`stack`" instead of "`temp`": temporary variables are not reentrant. After the `call fak1`, the values of `n`,`r`, and `n_1` are therefore undefined.

The next example shows the use of vectors. A function that computes a vector product is shown.

```
vector int(16) 100 (v1,v2) ;; all vectors are global

func v1_times_v2()
  temp index(v1,v2) i
  temp int(32) (sum,prod)
  sum = 0
  loop i (0,100,1)
    prod = "*" (v1[i],v2[i])
    sum = "+" (prod,sum)
  endloop
  return (sum)
```

The `loop` form is a built-in primitive with the three parameters "first index", "number of steps", and "step size". The variable `i` should be of type index or integer.

It is not possible to parameterize a vector-manipulating function like `v1_times_v2`, since there are no vector pointers. This may seem wasteful and overly restrictive, but has its reason. Especially in DSP processors, disjoint memory banks and hardware loop support are often found. A general `vector_prod` function could prove hard to realize on such chips, because vector sizes and alignments may have to be hard-coded into the loop instructions.

By disallowing general pointers to vectors, it is guaranteed that all vector index ranges can be computed and verified at assembly-time.

Vectors are accessed using a hard-wired subscript form. Subscripting is not modelled as a primitive, but as a built-in syntax form, since it is easier to expand a subscript operator that cannot be implemented as an addressing mode into a sequence of access computations than to do the converse.

9.3 *Structure of* M

The abstract data type of M types is:

```
datatype Mtype =
  Mt_int of int
| Mt_float of int * int
| Mt_index of string list
| Mt_bool
| Mt_string
```

M programs distinguish between local and global declarations. Functions and vectors can only be defined globally. A program consists of a series of global declarations (the order is not important):

```
datatype Mprogram =
  Mprog of Mfunc list * Mglob list
```

Besides functions, there are two kinds of global declarations: those for scalars (arbitrary atomic values), and those for vectors. Both kinds of declarations can declare a whole bunch of values of the same type.[4] Instead of a separate identifier type, the M abstract syntax employs simple strings.[5]

[4] The presence of multiple declarations simplifies some of the transformation rules that generate the M programs.

[5] Using strings instead of the symbols employed by the rest of the compiler makes it possible to separate any assembly-generation back-end totally from the rest of the compiler. As usual, the specification shown here exactly mirrors the current implementation.

```
datatype Mglob =
  Mscalar of string list * Mtype
| Mvector of string list * int * Mtype
```

A function declaration consists of a list of formal parameters, a set of local variable declarations, and a sequence of statements. The parameters and local variables have a scope and lifetime of the statements; it is not possible to declare function-local static variables.

```
datatype Mfunc =
  Mfunc of string * Mres list * Mdecl list * Mstmt list
```

Each `Mdecl` corrsponds to an invocation-local ("stack" or "auto") variable. When a function is called, its statements are executed sequentially. The two kinds of `Mdecl` differ in their stack behaviour: when a `Mapp` statement ("function call") is executed, the `Mstck` variables are saved on the stack, and restored upon return. The `Mtemp` variables are not saved; they are undefined after a function call.

```
datatype Mdecl =
  Mstck of string list * Mtype                       (* name, type *)
| Mtemp of string list * Mtype                       (* name, type *)
```

L-expressions (aka, locations) are modelled by `Mres` expressions. There are only two kinds of `Mres` expressions: variable names and vector locations. A variable name may only be used in a context in which it is defined (either via `Mdecl` or via `Mglob`). A vector location is denoted by the name of the vector and an arbitrary atomic expression (`Marg`) that computes to an index value of appropriate type.

```
datatype Mres =                                      (* result (l-expr) is: *)
  Mr_id of string                                    (* identifier *)
| Mr_index of string * Marg                          (* vector store *)
```

`Marg`s denote atomic values. Of the six `Marg` constructors, four denote literals; the other two denote the content of a variable (that has to be visible in the current context) and the content of a vector location. Note that vector access can be nested.[6]

```
datatype Marg =                                      (* argument (atomic expr) is: *)
  Mint of int                                        (* integer literal *)
| Mfloat of real                                     (* fp literal *)
| Mbool of bool                                      (* boolean literal *)
| Mstring of string                                  (* string literal *)
| Mid of string                                      (* content of a variable *)
| Mindex of string * Marg                            (* content of a vector element *)
```

Finally, there are statements. `Mapp` and `Mreturn` implement function call and return; `Mcall` invokes primitives; `Mjump` and `Mlabel` allow non-sequential control flow within a function (`Mjump` can pass arguments to the `Mlabel`); `Mif` is the standard `if` operator; `Mloop` is a hard-wired loop construct.

```
datatype Mstmt =                                     (* statement is: *)
  Mreturn of Marg list                               (* return (with results) *)
| Mapp of string * Marg list * Mres list             (* call a function (Mfunc) *)
| Mcall of string * Marg list * Mres list            (* call a primitive *)
| Mjump of string * Marg list                        (* goto label *)
| Mlabel of string * Mres list                       (* name & arguments passed *)
| Mif of Marg * Mstmt list * Mstmt list              (* if arg then ... else ... *)
| Mloop of string * Marg * Marg * Marg * Mstmt list  (* loopvar,init,#steps,incr *)
```

An invocation of `Mloop(var,init,steps,incr,statements)` corresponds to

[6] Allowing nested vector addressing such as `Mindex("a",Mindex("b",Mid("c")))` simplifies the grammar and cuts down the number of temporary variables in the generated M-code; it is trivial to atomize complex addressing modes when generating code for machines that do not provide such addressing modes.

```
var = init ; tmp = 0;
while tmp < steps do
  ...statements...
  var = var + incr;
  tmp = tmp + 1;
od
```

where `tmp` is an appropriate variable. This kind of loop description was chosen because it is most easily analyzed and transformed into equivalent forms.

9.4 *Transforming CF into* M

The transformations needed to generate M code from the reconstructed Code Form program are mostly trivial.

- *State Globalization*: One global variable is created for each distinct reference in the state, and reference lookups and definitions are translated in equivalent accesses to these globals. This can be done without further analysis because it is known that the state is single-threaded.
- *Tuple Atomization*: Non-atomic variables (i.e., tuples) are split up into sub-variables of atomic type.
- *Vector Globalization*: All Vector-typed variables are translated into global variables.

Of these points, only the last one is complicated. There are actually three transformations involved:

- *Introducing In-Place Primitives*: Some vector-creating primitives (`_update_vector`, `_map_vector`) are replaced by vector-modifying counterparts (`_update_vector_inplace`, `_map_vector_inplace`). This can be done in a correct way by first introducing copies:

  ```
  v2 = updating_primitive(...,v1,...)
  ```

  will be transformed into

  ```
  v2 = copy_vector(v1)
  v2 = updating_primitive_inplace(...,v2,...)
  ```

- *Dead-Vector Elimination*: Whenever a vector variable becomes unused, its last copy operation can be eliminated:

  ```
  v2 = copy_vector(v1) \\ v1 unused after this
  v2 = updating_primitive_inplace(...,v2,...)
  ```

  can be replaced by

  ```
  v2 = updating_primitive_inplace(...,v1,...)
  ```

- *Removing Vector-Typed Function Arguments*: Functions that take vectors as arguments are copied; a different instance is created for every vector they might be applied to.[7] When this transformation has been performed successfully, all vectors are effectively single-threaded, and can be made global.

  Finally, all occurences `_lookup_vector` and `_update_vector_inplace` are replaced by their M counterparts (`Mindex` and `Mrindex`), and calls to `_map_vector_inplace` are replaced by `Mloop` invocations.

Much energy can be spent into providing further optimizations to minimize the number of "scratch vectors". The current `ac` only implements the most basic heuristic: it tries to move all non-modifying accesses to a vector before the first modifying access.

[7] In typical DSP code, this should not pose any code-size problems, since such applications usually employ a small number of fixed vectors. The current CF-to-M translator uses a primitive `clone_vector` that creates a dynamic, garbage-collected copy of a vector for functions that introduce vectors on the fly, or within recursive functions. The `C` backend might implement this with `malloc` and reference counting.

9.5 The C back-end

The only back-end currently implemented for M generates C code. The resultant code is quite inefficient, since C is not especially suited as a low-level assembler:

- There is no possibility to get fine control over stack allocation behaviour. Thus, the `temp` variables employed in M cannot be correctly represented.
- C functions cannot have multiple output values. It is possible to use structures for this purpose, but those are not held in registers by most compilers.

The M-to-C translater generates one big `switch` statement, in which labels and function heads are represented as `case`s. The stack is held explicitly, and the arguments that are passed along by each `jump` or `call` are modelled via global transfer registers.

9.6 The Future: Automatic Back-End Generation

A more ambitious approach to machine code generation is the automatic generation of back-ends. Languages like *nML* [43, 44] give a full instruction-level register-transfer semantics for a target architecture, together with information about the assembly language syntax and bit-level code representation. In a framework like CBC [38, 39], such a target description is transformed into a hardware-model suited as input language for a CDFG (Control/Data-Flow Graph) code generator. By providing CDFG output, `ac` can be retargetted to arbitrary target architectures. It is also possible to transform the description into a simulation tool [38], which allows target-instruction level debugging of programs.

Chapter 10

Conclusions and Outlook

The goal of this work was to establish a compilation technique able to cope with "rich" languages like ALDiSP, which have both a complex semantics and stringent requirements in regard to space and time efficiency, but which do not have to be compiled with a fast turn-around time, since actual software development occurs in an interpreter environment, and cross-compilation is the order.

ALDiSP is un-compilable with the methods used to compile other classes of languages:

- *Third-Generation Languages* (Fortran, Pascal, Modula-2, C, C++) are based on the assumption that each source language construct can be modelled effectively by a (small) chunk of machine code. The data types provided as primitive and operations on them are essentially the atomic types and operations of the target hardware (numbers, characters, addresses). Code generation for such languages can be achieved by mapping each construct to its target code sequence; optimization mostly consists of register allocation and instruction scheduling.
- *Functional and Logic Programming Languages* (Haskell, SML, Prolog) are based upon abstract machines and runtime systems that implement efficient garbage collected heaps for many small objects. Code generation consists of creating efficient code sequences that work as new operators (supercombinators) of these abstract machines. Typical applications of such languages need to represent and work upon large and complex heterogenous data structures (list, graphs, knowledge bases). Dynamically typed languages (LISP, Scheme) are often not compiled down to the level of machine data types ("unboxed values"), but only to that of the abstract machine ("boxed values") so that some degree of interpretation takes place at run-time.
- *Signal-Flow Languages* (Silage, Signal) essentially describe arithmetic expressions, and can be compiled as basic blocks: each signal value can be represented as a variable, and each operation as a hardware instruction. The only optimizations left to the compiler are register allocation, memory layout, and instruction scheduling.

As a language, ALDiSP violates the assumptions behind the aforementioned compilation methods:

- The control flow is not made explicit, but emerges from the interaction of data types (vectors, streams, overloaded functions) and generic rules (automapping, overload resolution, blocking and forcing).
- ALDiSP has no static type system.
- The target architectures for which ALDiSP is intended do not provide large memories, and run-time garbage collection is out of the question.

- ALDiSP supports data types and operations on them that are not primitive machine types and operations (whole arrays, higher-order functions).

The DSP algorithms to which ALDiSP is typically applied are, however, simple and highly regular. The idea behind `ac` was therefore to write a compiler that filtered this specific regular structure out of ALDiSP programs that are composed from very generic building blocks. Online partial evaluation promised to provide a method for this.

10.1 Development History

Early work on `ac` was guided by a paper [121] describing *Fuse*, an online partial evaluator for Scheme. The first attempts at writing `ac` were based on extensions to a naïve ALDiSP interpreter written in Scheme. While Scheme is a fine language for experimental programming, language extensions and meta-interpreters, it lacks in some respects. Especially the need for static type checking motivated the complete re-write into SML.

In parallel to the compiler, a first complete semantics was written. An earlier attempt to write a "direct" ALDiSP semantics didn't succeed, since the semantics got too complex. Thus, the Lambda Form was born. In retrospect, the Lambda Form is *still* too complex; it might have been better to transform ALDiSP directly into what is now the Code Form. At that time, I considered such a decomposition, but was afraid that the intermediate code size would explode if the Lambda Form got too basic.

The basic idea of Fuse, namely the graph-based representation, was mostly lost in `ac`, since ALDiSP features such as the suspension state and full higher-order functions are not considered by Fuse, and cause quite some problems when a "flat" program is to be reconstructed from a graph. The major conceptual breakthrough in the reconstruction phase consisted of the idea to separate the `code` and `atom` attributes, and to use the abstract state graph as the skeleton of the reconstructed program. The Code Form language changed a lot over the time; it was originally a strict subset of the Lambda Form.

The current implementation of `ac` is not very stable and cannot be used to compile large programs, since its resource consumption is too high. As a proof-of-concept, it shows that ALDiSP *can* be compiled into efficent code. It is probably impossible to speed up the compiler by a significant amount without redesigning it completely; a supercombinator-based compiler that has a powerful abstract graph reduction machine as its target (cf. section 10.2.3) combined with a partial evaluator for said machine appears to be a viable approach.

10.2 Design Alternatives

There are some major and many minor design alternatives that might have been chosen during the development of the ALDiSP compiler. Some of them occurred to me too late, others involved too many unknown factors.

10.2.1 Explicit State Passing

A central part of the current compiler is the abstract scheduler, which tries to enumerate all possible "states" the program under compilation can encounter. If this scheduler fails, no compilation is possible. If it succeeds, garbage collection can be done at compile-time, and a fixed memory-layout is guaranteed.

Many real programs have a very large state space, or no static schedule at all (i.e., an infinite state space). Programs that create dynamic data structures whose "form" depends upon the input fall under this category;

"form" here refers to the comparison function employed by the abstract scheduler that tells whether two states can be mapped together; to have a similar form, two data structures must have the same structure and size.

The current compiler does not treat states as first-class values and part of the domain hierarchy on which the abstract interpreter is based. As a consequence, there are no "fully abstract" states, only states that may contain references with abstract definitions. It might be interesting to remove the abstract scheduler as a separate entity by transforming the intermediate representation into a form that explicitly passes the state around.

Such an approach would however need a more sophisticated loop detection scheme, since the abstract interpreter would now have the additional task of creating the "state automaton".

10.2.2 Continuation-Passing Style

Early in the development of the ALDiSP semantics and compiler, I considered a continuation-passing style (CPS) intermediate form. Such a form goes beyond explicit state passing, in that it also makes *control flow* explicit. CPS is not based upon the "function call stack" paradigm of flow control, but on explicit continuation functions which provide return addresses and stack frames. CPS would have simplified the modelling of exceptions to some degree. I abandoned this approach because I felt uneasy with an intermediate representation that involves the heavy use of higher-order functions. Abstract interpreters, especially the loop detection heuristics, have problems with higher-order function arguments, since the "similarity" of such is hard to determine. It is much easier to inspect a classical call stack, since terms like "call depth" are naturally expressed in such a framework, whereas they are, in a CPS setting, structural properties of anonymous functions. My experiments with CPS representation also generated intermediate programs that were exceedingly hard to understand, a property that is not at all wished for in an experimental compiler.

10.2.3 Compilation by Graph Reduction

An entirely different approach to the whole compilation process is based upon the *graph reduction* model [116]. Graph reduction is based upon an rewriting engine that reduces a program graph into a *normal form*. The graph is a representation of combinators; any functional program can be translated into a set of combinators by a process called bracket abstraction. Both program and data have a common form; the "result" of a program is its normal form. Graph reduction is especially suited for *lazy* languages, since there are no "pure data" nodes; the simplest node is the "quote" or "identity" node that corresponds to the `Lit` of LF. A strict primitive function is reducible if all its input arcs are connected to these identity nodes[1]. This can be exploited by lazy languages, in which argument passing is done by-name; in ALDiSP terms, every expression is wrapped by a `delay`. Due to this characteristic, each access to a value must first check its value-ness, i.e. whether it has already been forced or not. By providing "identity" nodes, this information is encoded in the value's node type (an unevaluated value is a non-identity node).

While ALDiSP is not *per se* lazy, it does share some properties with lazy languages. This is due to the similarity of suspensions and promises; i.e. to the "blocking" and "forcing". The graph-reduction correspondence to blocking is the inability to evaluate a graph while its arguments are not yet in normal form; the equivalent to forcing is the standard action of graph reduction – in graph reduction, an argument's value is accessed by forcing its node. The most important optimization for lazy languages is *strictness analysis*; an argument to a function is strict if its value is accessed in every possible evaluation of the function. Strict arguments can be passed by-value. Finding strict function arguments amounts to finding "connected groups" of expressions; if any member of such a group has to be evaluated, all members have to be evaluated. In ALDiSP strictness

[1]And if they are of the right type – all graph-reduction systems I know of work only on statically well-typed programs, but this is mostly for speed-up.

analysis can be used to force promises and, more important, to block a function application when one of the arguments is an unevaluated suspension.

Th alternative approach to ALDiSP compilation would use a new semantics, built from quite a different set of primitives. If the "abstract machine" provides for automatic blocking and forcing, the most onerous parts of ALDiSP, namely the details related to promises and suspensions, would be delegated to the abstract machine, which would also provide for memory allocation and garbage collection. The compiler would consist of a fairly standard type inference system, or a primitive abstract interpretation that ignores delays and promises, and tries to find the type of every function application so as to resolve auto-mapping and overloading at compile time. The compiled program would be a set of type-correct supercombinators, which could be translated one-to-one to graph code. The graph reduction machine would correspond to one of the standard architectures, e.g. the G-machine [69] and its derivates [65, 24, 37, 6, 11], which has to be extended by an I/O and time manager that corresponds to, from the machine's perspective, a non-deterministic evaluator of suspension nodes.

Such a graph reduction ALDiSP would have a very poor performance when compared with the output of `ac`, especially when memory consumption is considered. The trick would be to perform a partial evaluation of the graph reduction engine and the program (or parts of it). In a graph context, the problems of program representation (cf. chapter 8.8) would be minimized; also, some optimizations due to partial evaluation could be applied even to those programs that cannot be compiled by `ac` because they have no static schedule.

10.3 *Redesigning* ALDiSP

The experiences made during the implementation of the ALDiSP compiler showed up some weaknesses in the language, and provided the motivation for the design of a successor language, AL-2. AL-2 is designed with reasonably fast and simple compilation in mind; it is described in detail in [47]. Here is the list of major differences to ALDiSP:

- AL-2 was co-designed with its *semantics*, which therefore is much simpler and smaller than ALDiSP's.
- AL-2 allows *static typing*. An advanced type system akin to Haskell's [58] *type classes* [119] provides most of the freedom available to ALDiSP's numerical expressions, while giving much better protection against unreasonable type errors.
- A specialized *signal type* is introduced. A signal is a sequence of data objects, each of which has associated timing information. Signals replace ALDiSP's streams and pipes. A set of synchronization primitives on signals replace most uses of the `suspend` and `delay` constructs. Most of the run-time complexity associated with promises and suspensions essentially vanishes; "promise returning α" and "suspension returning α" become compile-time types.
- AL-2 is specified as a *core language* with *extensions*. The core language can be implemented without any sophisticated compilation techniques; most of the extensions can be realized as preprocessors that generate core language programs.
- Amongst the extensions are the *module system* and the *macro processor*. The macros can be typed.
- As an extension to the type system, *equational assertions* replace the predicate types of ALDiSP.

Once a working AL-2 compiler exists, it might be interesting to write a converter that transforms old ALDiSP programs into AL-2.

Bibliography

[1] S. Abramsky, C. Hankin (eds): *Abstract Interpretation of Declarative Languages*, Ellis Norwood Ltd, 1987

[2] ACM: *Proc. Third Workshop on the Mathematical Foundations of Programming Languages Semantics*, Springer LNCS 298, April 1987

[3] F. Allen, J. Cocke, *A catalogue of optimizing transformations*, in: R. Rustin (ed.): Design and Optimization of Compilers, Prentice-Hall, 1972

[4] A. W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992

[5] A. W. Appel, T. Jim, *Continuation-passing, closure-passing style*, in: [97], pp. 293-302

[6] G. Argo, *Improving the Three Instruction Machine*, in: [51]

[7] J. Armstrong, R. Virding, M. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1993

[8] J.L. Armstrong, B.O. Däcker, S.R. Virding, M.C. Williams, *Implementing a Functional Language for Highly Parallel Real Time Applications*, published in SETSS 92, 30.3.-1.4.1992, Florence; available via `ftp://euagate.eua.ericsson.se/pub/eua/erlang/info/implem.ps.Z`

[9] E.A. Ashcroft, *Lucid, a Nonprocedural Language with Iteration*, in: CACM 20, No. 7, July 1977, pp. 519-526

[10] L. Beckman, A. Haraldson, Ö. Oskarsson, E. Sandewall, *A Partial Evaluator and its Use as a Programming Tool*, in: Artificial Intelligence Journal 7 (1976), pp. 319-357

[11] P. Bellot, *Graal: A functional programming system with uncurryfied combinators and its reduction machine*, in: [34]

[12] A. Benveniste, P. Bournai, T. Gautier, P. Le Guernic, *SIGNAL: A Data Flow Oriented Language for Signal Processing*, INRIA Rapports de Recherche No. 378, INRIA Centre de Rennes IRISA

[13] A. Berlin, *A Compilation Strategy for Numerical Programs Based on Partial Evaluation*, MIT Technical Report AI-TR 1144, 78 pp.; MIT AI Lab, 1989

[14] A. Berlin, D. Weise, *Compiling Scientific Code Using Partial Evaluation*, in: IEEE Computer, Vol. 23, No. 12, Dec 1990, pp. pp. 25-37

[15] V. Berzins, *Semantics of a Real-Time Language*, in: Proc. IEEE Symp. on Real-Time Systems, Dec. 1988, pp. 106-110

[16] V. Berzins, *Execution of a High Level Real-Time Language*, in: Proc. IEEE Symp. on Real-Time Systems, Dec. 1988, pp. 69-76

[17] D. Bjœrner, A.P. Ershov, N.D. Jones, *Partial Evaluation and Mixed Computation*, Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, 18-24 Oct. 1987, North-Holland, 1988

[18] A. Bloss, *Update Analysis and the Efficient Implementation of Functional Aggregates*, in: [51]

[19] A. Bloss, *Path Analysis and the Optimization of Non-Strict Functional Languages*, Ph.D. thesis, Yale University, Dept. of Computer Science, 1989. Available as research report YALEU/DCS/RR-704

[20] A. Bloss, P. Hudak, *Path Semantics*, in: [2]

[21] A. Bloss, P. Hudak, J. Young, *An optimizing compiler for a modern functional langauge*, in: *The Computer Journal*, Vol. 31, No. 6, pp. 152-161, 1988

[22] R.M. Burstall, J. Darlington, *Some transformations for developing recursive programs*, in: [94]

[23] R.M. Burstall, J. Darlington, *A transformation system for developing recursive programs*, in: Journal of the ACM, 24(1), pp. 44-67, January 1977

[24] L. Cardelli, *The Amber Machine*, in: [25], pp. 48-70

[25] G. Cousineau, P.-L. Curien, B. Robinet, *Combinators and Functional Programming*, Springer LNCS 242

[26] P. Caspi, N. Halbwachs, D. Pilaud, J.A. Plaice, *LUSTRE, a declarative language for programming synchronous systems*, in: [96] and as Report L-1, Project SPECTRE Groupe Speécification et Analyse des Systemès, Laboratoire de Génie Informatique de Grenoble

[27] W. Clinger, J. Rees (eds.), *Revised*^{3} *Report on the Algorithmic Language Scheme*, in: SIGPLAN Notices, 21(12), Dec. 1986 and as MIT AI Memo 848a, Sept. 1986

[28] W. Clinger, J. Rees (eds.), *Revised*^{4} *Report on the Algorithmic Language Scheme*, University of Oregon Technical Report CIS-TR-90-02

[29] C. Consel, O. Danvy, *From Interpreting to Compiling Binding Times*, in: [35]

[30] C. Consel, S.C. Khoo, *Parameterized Partial Evaluation*, in: ACM Trans. on Programming Languages and Systems, Vol. 15, No. 3, July 1993, pp. 463-493

[31] J. Darlington, *A semantic approach to automatic program improvement*, Experimental Programming Reports: No. 27, School of Artificial Intelligence, University of Edinburgh, 1972

[32] J. Darlington, R.M. Burstall, *A system which automatically improves programs*, in: Proc. of the Third International Joint Conference on Artificial Intelligence, Stanford, Calif., 1973

[33] A.P. Ershov, V.V. Grushetsky, *An Implementation-Oriented Method for Describing Algorithmic Languages*, in: B. Gilchrist (ed.): Information Processing 77, Toronto, Canada; North-Holland, 1977; pp. 117-122

[34] B. Robinet, R. Williams (eds): *ESOP '86 – 1st European Symposium on Programming*, Springer LNCS 213

[35] —EN. Jones (ed) *ESOP '90 – 3rd European Symposium on Programming*, Springer LNCS 432

[36] B. Krieg-Brückner (ed.), *ESOP '92 – 4th European Symposium on Programming*, Rennes, France, February 1992, Springer: LNCS 582

[37] J. Fairbairn, S. Wray, *Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators*, in: [50], pp. 34-44

[38] A. Fauth, M. Freericks, A. Knoll, *Generation of Hardware Machine Models from Instruction Set Descriptions*, in: VLSI Signal Processing VI, Eggermont et.al. (eds), IEEE Signal Processing Society, 1993

[39] A. Fauth, *The CBC Compiler Generator: The Final Report*, Forschungsberichte des Fachbereichs Informatik Nr. 1994/13, TU Berlin

[40] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen, *The Essence of Compiling with Continuations*, in: [100]

[41] M. Freericks, *Entwurf und Spezifikation einer funktionalen Programmiersprache für die speziellen Erfordernisse der digitalen Signalverarbeitung*, Diplomarbeit, TU Berlin, 1990

[42] M. Freericks, A. Knoll, *ALDiSP - eine applikative Programmiersprache für Anwendungen in der digitalen Signalverarbeitung*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/9, TU Berlin

[43] M. Freericks, *The nML Machine Description Formalism*, Forschungsberichte des Fachbereichs Informatik Nr. 91-15, TU Berlin, 1991

[44] M. Freericks, *The nML Machine Description Formalism (updated Version 1.2)*, in: ESPRIT-II Project 2260 SPRITE Progress Report for Period June 1992-November 1992, Report No. PR-4.2, 1. Dec. 1992, Editor: Patrick Pype

[45] M. Freericks, A. Knoll, L. Dooley, *The Real-Time Programming Language ALDiSP-0: Informal Introduction and Formal Semantics*, Forschungsberichte des Fachbereichs Informatik Nr. 92-26

[46] M. Freericks, A. Fauth, A. Knoll, *A Basic Semantics for Computer Arithmetic*, Technischer Bericht 94/6, Technische Fakultät, Universität Bielefeld

[47] M. Freericks, A. Knoll, *The AL-2 Signal Processing Language: A Successor to ALDiSP*, Technischer Bericht, Technische Fakultät, Universität Bielefeld (to appear)

[48] Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, *Essentials of Programming Languages*, MIT Press, 1992

[49] J.-P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture*, Nancy, France, 1985; Springer, LNCS 201

[50] *Proc. IFIP Symposium on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987

[51] ACM SIGPLAN,/SIGACT, IFIP, *Functional Programming Languages and Computer Architecture*, Imperial College, London, Sept. 11-13, 1989

[52] Y. Futamura, *Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler*, in: Systems, Computers, Controls, 2(5), 1971, pp. 45-50

[53] Y. Futamura, K. Nogi, *Generalized Partial Computation*, in: [17], pp. 133-152

[54] T. Gautier, P. Le Guernic, O. Maffeïs, *For a New Real-Time Methodology*, IRISA Publication Interne No. 870, October 1994

[55] N. Halbwachs, D. Pilaud, F. Ouabdesselam, A-C. Glory, *Specifying, Programming and Verifying Real-Time Systems using a Synchronous Declarative Language*, Report L-6, Project SPECTRE Groupe Speécification et Analyse des Systemès, Laboratoire de Génie Informatique de Grenoble

[56] A. Haraldsson, *A program manipulation system based on partial evaluation*, Ph.D. thesis, Linköping Univerity, Sweden, 1977., Linköping Studies in Science and Technology Dissertations 14

[57] P.G. Harrison, *Function Inversion*, in: [17], pp. 153-166

[58] P. Hudak, S.P. Jones, P. Wadler (eds): *Report on the Programming Language Haskell (version 1.2)*, 1st March 1992, available via `ftp` from numerous sites

[59] P. Hudak, *A semantic model of reference counting and its abstraction*, in: [1]

[60] J. Hughes, *Analysing Strictness by Abstract Interpretation of Continuations*, in: [1]

[61] J. Hughes, *Backwards Analysis of Functional Programs*,, in: [17] pp. 187-209

[62] Institute of Electrical and Electronics Engineers, Inc., *Draft Standard for the Scheme Programming Language*, P1178/D4, March 7, 1990

[63] T.P. Jensen, T. Æ. Mogensen, *A Backwards Analysis for Compile-Time Garbage Collection*, in: [35], pp. 227-239

[64] T. Johnsson, *Lambda Lifting: Transforming Programs to Recursive Equations*, in: [49], pp. 190-203

[65] S. L. Peyton Jones, J. Salkind, *The Spineless Tagless G-Machine*, in: [51]

[66] S.P. Jones and C. Clack, *Finding fixpoints in abstract interpretation*, in: [1], pp. 246-265

[67] Neil D. Jones, Carsten K. Gomard, Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993

[68] D. Keppel, S. J. Eggers, R. R. Henry, *A Case for Runtime Code Generation*, University of Washington Department of Computer Science and Engineering, UWCSE 91-11-04, November 1991

[69] R.B. Kieburtz, *The G-machine: A fast, graph-reduction evaluator*, in: [49], pp. 400-413

[70] C.D. Kloos, *STREAM: A Scheme Language for Formally Describing Digital Circuits*, in: PARLE – Parallel Architectures and Languages Europe, Vol. II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987

[71] Alois Knoll, Markus Freericks, *An applicative real-time language for DSP-programming supporting asynchronous data-flow concepts*, in: Microprocessing and Microprogramming, Vol. 32, No. 1-5 (August 1991 - Proceedings Euromicro '91), pp. 541-548

[72] A. Knoll, A. Schweikard, M. Freericks, *Eine datenflußorientierte, funktionale Programmiersprache für die Echtzeitdatenverarbeitung*, in: Prozeßrechensysteme '91, Proceedings, Belin, Februar 1991 Springer Informatik-Fachberichte 269

[73] Alois Knoll, Rupert C. Nieberle, *CADiSP – A Graphical Compiler for the Programming of DSP in a Completely Symbolic Way*, Proc. Int. Conf. on Acoustics, Speech and Signal Processing, April 1990

[74] E.E. Kohlbecker Jr, *Syntactic Extensions in the Programming Language Lisp*, Ph.D. thesis, Indiana University, August 1986

[75] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, *ORBIT: An optimizing compiler for Scheme*, Proc. Sigplan '86 Symp. on Compiler Construction, published as: SIGPLAN Notices 21(7), pp. 219-233, July 1986

[76] V. Kruckemeyer, A. Knoll, Eine imperative Sprache zur Programmierung digitaler Signalprozessoren, Forschungsberichte des Fachbereichs Informatik Nr. 1990/10, TU Berlin

[77] L. Lamport, *What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority*, in: [95], pp. 78-83

[78] P. Lee, *Realistic Compiler Generation*, Cambridge, MA: MIT Press, 1989

[79] *Proc. of the 1990 Conference on Lisp and Functional Programming*, Nice, France, June 1990

[80] F. Löhr, A. Fauth, M. Freericks, *SIGH/SIM - An Environment for Retargetable Instruction Set Simulation*, Forschungsberichte des Fachbereichs Informatik Nr. 93-43, TU Berlin, 1993

[81] L.A. Lombardi,B. Raphael, *Lisp as the Language for an Incremental Computer*, in: E.C. Berkeley and D.G. Bobrow (eds.): *The Programming Language Lisp: Its Operation and Applications*; MIT Press, Cambridge, Massachusetts, 1964; pp. 204-219

[82] D.B. Loveman, *Program improvements by source to source transformations*, in: [94]

[83] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, M.I. Levin, *LISP 1.5 Programmer's Manual*, The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, MIT Press, 1965

[84] J. McGraw, *Parallel Functional Programming in Sisal: Fictions, Facts, and Future*, UCRL-LC-114360, June 1993

[85] R. McConnell, *Prototyping of VLSI Components from a Formal Specification*, IRISA Publication Interne No. 865, Septembre 1994

[86] U. Meyer, *Techniques for Partial Evaluation of Imperative Languages*, in: [113], pp. 94-105

[87] R. Milner, M. Tofte, R.Harper, *The Definition of Standard ML*, MIT Press, 1990

[88] R. Milner, M. Tofte, *Commentary on Standard ML*, MIT Press, 1991

[89] T. Mogensen, *Binding time aspects of partial evaluation*, Ph.D. thesis, DIKU, University of Copenhagen, Denmark, March 1989

[90] Joel Moses, *The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem*, in: 2nd Symposium on Symbolic and Algebraic Manipulation, Los Angeles, 1970; published in SIGSAM Bulletin No. 15, July 1970, pp. 13-27

[91] Hanne Riis Nielson, Flemming Nielson, *Bounded Fixed Point Iteration (Extended Abstract)*, in: [98], pp. 71-82

[92] V. Nirkhe, W. Pugh, *Partial Evaluation of High-Level Imperative Programming Languages with Applications in Hard Real-Time Systems*, in: [98], pp. 269-280

[93] Alan V. Oppenheim, Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989

[94] ACM, *3rd ACM Symposium on Principles of Programming Languages (POPL'76)*, Atlanta, Georgia; January 19-21, 1976

[95] ACM, *12th ACM Symposium on Principles of Programming Languages (POPL'85)*, New Orleans, Lousiana; January 14-16, 1985

[96] ACM, *14th ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, Germany; January 21-23, 1987

[97] ACM, *16th Conf. on Principles of Programming Languages (POPL'89)*, Austin, Texas; January 11-13, 1989

[98] ACM, *19th Conf. on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico; January 19-22, 1989

[99] ACM, *21st Conf. on Principles of Programming Languages (POPL'94)*, Portland, Oregon; January 1994

[100] ACM, *Programming Language Design and Implementations (PLDI' 93)*, Albuquerqe, NM; June 23-25, 1993; published as SIGPLAN Notices Vol. 28, No. 6, June 1993

[101] H. John Reekie, *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*, (PhD thesis) School of Electrical Engineering, University of Technology, Sydney; 1995

[102] E. Ruf, D. Weise, *Opportunities for online partial evaluation*, Stanford University, California, April 1992; technical report CSL-TR-92-512

[103] E. Ruf, D. Weise, *Preserving Information during Onling Partial Evaluation*, Stanford University, California, April 1992; technical report CSL-TR-92-517

[104] E. Ruf, *Topics in online partial evaluation*, Ph.D. thesis, Stanford University, California, February 1993; published as technical report CSL-TR-93-563

[105] P.B. Schenk, E. Angel, *A FORTRAN to FORTRAN optimizing compiler*, in: The Computer Journal, Vol. 16, No. 4, 1972

[106] D. A. Schmidt, *Denotational Semantics*, Allyn and Bacon Inc, 1986

[107] E. Schoen:, *The CAOS System*, Stanford University Report No. STAN-CS-86-1125, 1986

[108] A. Schwarte, H. Hanselmann, *The Programming Language DSPL*, PCIM'90, München

[109] P. Sestoft, *Replacing Function Parameters by Global Variables*, in: [51], pp. 39-53

[110] *EDC/Silage Reference Manual*, European Development Center, 1989

[111] *Silage User's and Reference Manual*, prepared by Mentor Graphics/EDC, June 1991, (describes version 2.0)

[112] Guy L. Steele, jr., *Rabbit: a compiler for Scheme*, Tech. Report AI-TR-474, MIT, Cambridge, 1978

[113] *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale Univ., June 17-19, 1991, published as: SIGPLAN Notices, Vol. 26, No. 9, Sept. 1991

[114] V.F. Turchin, *The concept of a supercompiler*, in: ACM Transactions on Programming Languages and Systems, 8(3), July 1986, pp. 292-325

[115] V.F. Turchin, *The algorithm of generalization in the supercompiler*, in: [17], pp. 531-549

[116] D.A. Turner, *A New Implementation Technique for Applicative Languages*, Software–Practice and Experience, Vol. 9(1979), pp. 31-49

[117] P. Wadler:, *Comprehending Monads*, LFP'90 [79], pp. 61-78

[118] P. Wadler, J. Hughes, *Projections for Strictness Analysis*, in: [50]

[119] P. Wadler, S. Blott, *How to make ad-hoc polymoprhism less ad-hoc*, in: [97]

[120] D. Weise, *Graphs as Intermediate Representations for Partial Evaluation*, Stanford University, California, March 1990, technical report CSL-TR-90-421

[121] D. Weise, R. Conybeare, R. Ruf, S. Seligman, *Automatic online partial evaluation*, in: Symposium on Partial Evaluation and Semantics-Based Program Manipulation (New Haven, Conn., June 1991), pp. 1-11

[122] D. Weise, E. Ruf, *Computing types during program specialization*, Stanford University, California, August 1990, technical report CSL-TR-90-441

[123] D. Weise, R.F. Crew, M. Ernst, B. Steensgaard, *Value Depence Graphs: Representation without Taxation*, Microsoft Research, available via `ftp://research.microsoft.com/pub/papers/vdg.ps`, 1994; revised version of a paper that appeared in [99], pp. 297-310

[124] W. Wulf, R.K. Johnson, C.B. Weinstock, S.O. Hobbs, C.M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, 1975

Index

`is`, 56

abstract
    reference counts, 5

CADiSP, 2
CPS, 4

details
    implementation specific, 2
DSP
    applications of, 3

kernel
    of ALDiSP, 2

open language
    why ALDiSP is one, 2
order of execution, 4

piece de resistance, 6
predefined
    `Obj`, 68
    `false`, 68
    `is`, 56
    `true`, 68
    `_cast`, 63, 68
    `_checkfunctype`, 60, 68
    `_closure`, 56, 68
    `_delay`, 64, 68
    `_is`, 68
    `_overload`, 51–53, 68
    `_return`, 64, 68
    `_suspend`, 64, 68
    `_testfunctype`, 62, 68
    `_tupel`, 61, 63, 68
    `_unit`, 65, 68

static
    memory layout, 4

tail recursivity, 4