HUSC Language and Type System

By
Gordon J. Uszkay, B.Math.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science
Department of Computing and Software
McMaster University

© Copyright by Gordon J. Uszkay, June 1, 2006

MASTER OF APPLIED SCIENCE(2006) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: HUSC Language and Type System

AUTHOR: Gordon J. Uszkay, B.Math.(University of Waterloo)

SUPERVISOR: Dr. Christopher Kumar Anand and Dr. Jacques Carette

NUMBER OF PAGES: 1, 281

Abstract

HUSC is a high level declarative language designed to capture the relations and properties of a complex system independently of implementation and platform. It is intended for use in Coconut (COde CONstructing User Tool), a project at McMaster University to create a new development environment for safety-critical, computationally intensive domains such as medical imaging. The language is intended to provide an interface that is comfortable for use by scientists and engineers, while providing the benefits of strong, static type analysis found in functional programming languages.

HUSC supports type inferencing using constraint handling rules, including both predefined, parameterized shapes and an arbitrary number of additional properties or constraints called attributes. Each attribute class provides its own type inferencing rules according to the HUSC attribute class definition structure. Multiple implementations of an operator or function can each specify a specialized type context, including attributes, and be selected based on type inferencing. The HUSC system creates a typed code hypergraph, with terms as nodes and edges being all of the operator or functions implementations that are satisfied in the type context, which can be used as the basis for an optimizing compiler back end.

We present here the HUSC language and type system, a prototype implementation and an example demonstrating how the type inferencing may be used in conjunction with the function joins to provide a good starting point for code graph optimization. It also includes a number of observations and suggestions for making the transition from prototype to useable system. The results of this work are sufficient to demonstrate that this approach is promising, while highlighting some difficulties in producing a robust, practical implementation.

Acknowledgements

First, I would like to thank my supervisors, Drs. Jacques Carette and Christopher Kumar Anand, for their inspiration, supervision, support and guidance.

I would like to thank my committee members, Drs. Franya Franek and William Farmer for their patience and assistance, and Dr. Franek again for his guidance and encouragement early in the project. I would also like to thank Dr. Wolfram Kahl for his assistance on the project and instruction in the art of functional programming.

I also owe a big thanks to my family for their encouragement, sacrifice and support during some difficult times.

Contents

1 Introduction **1**
1.1 Document Structure . . . 2
1.2 Coconut Project . . . 4
1.3 Example: LA Pack in HUSC . . . 7
1.3.1 Literate HUSC Example . . . 9
1.4 HUSC — High level User Specification of Constraints . . . 13
1.4.1 Domain specific declarative programming model . . . 13
1.4.2 Join Expressions . . . 14
1.4.3 Shapes and Attributes . . . 15
1.4.4 Type Inferencing . . . 15
1.4.5 Assertions, Tests and External Proof Obligations . . . 17
1.5 Implementation Overview . . . 18
1.6 Summary . . . 19

2 Background Research **21**
2.1 Mathematics Programming Languages . . . 21
2.1.1 APL, J . . . 21
2.1.2 MatLab . . . 23
2.1.3 Mercury . . . 24
2.2 Type Theory and Type Systems . . . 24
2.2.1 Constraint Based Type Systems . . . 25
2.2.2 Constraint Handling Rules . . . 26
2.2.3 FISh — Static Shape Analysis . . . 27
2.2.4 Types containing Proofs . . . 27
2.2.5 Background Summary . . . 28

3 HUSC Language Definition: Syntax and Semantics **30**
3.1 Abstract Syntax . . . 30
3.1.1 Modules . . . 31
3.1.2 Module Imports . . . 31

3.1.3 Data Definitions and Declarations . . . 32
3.1.4 Operator Definitions . . . 33
3.1.5 Joined and Conditional Expressions . . . 33
3.1.6 Expressions . . . 34
3.1.7 Factors, SubExpressions . . . 34
3.1.8 Types . . . 35
3.1.9 Attribute Classes . . . 36
3.2 Semantics . . . 37
3.2.1 Programs . . . 39
3.3 Modules . . . 39
3.3.1 Module Bodies . . . 39
3.3.2 Module Context . . . 40
3.4 Declarations . . . 42
3.5 Definitions . . . 43
3.5.1 Function Definitions . . . 43
3.5.2 Partial Definition . . . 44
3.5.3 Total Definitions . . . 44
3.6 Operator Definitions . . . 45
3.6.1 Operator Symbol Table . . . 45
3.7 Importing Modules . . . 45
3.8 Expressions . . . 45
3.8.1 Simple Expressions . . . 46
3.8.2 Joined Expressions . . . 47
3.8.3 Local Body and Local Context . . . 49

4 Concrete Syntax 51
4.1 Literate HUSC . . . 51
4.2 Informal Description and Examples . . . 52
4.2.1 Declarations . . . 52
4.2.2 Module Import Interface . . . 52
4.2.3 Definitions and Expressions . . . 53
4.2.4 Joined Expressions . . . 54
4.2.5 Conditional Expressions . . . 55
4.2.6 Types . . . 55
4.2.7 Attributes . . . 57
4.2.8 Attribute Class Definition . . . 57
4.2.9 Operator Definition . . . 57

5 HUSC Type System **59**
5.1 Types . . . 59
5.1.1 Shapes . . . 59
5.1.2 Attributes . . . 64
5.1.3 Named Types and the Subtype Hierarchy . . . 65
5.2 Subtyping . . . 66
5.2.1 SubShape . . . 67
5.2.2 Attribute Subtyping . . . 68
5.3 Joins and Meets . . . 68
5.3.1 Joins and Meets on Shapes . . . 69
5.3.2 Joins and Meets on Attributes . . . 70
5.3.3 Joins and Meets for Named Types . . . 70
5.4 Attribute Classes . . . 70
5.4.1 Attribute Context . . . 71
5.4.2 Attribute Class Interpretation . . . 71
5.5 Type Judgements . . . 73
5.5.1 Overview . . . 73
5.5.2 Declarations and Definitions . . . 74
5.5.3 Function Declaration and Definitions . . . 74
5.5.4 Operator Declaration and Definition . . . 75
5.6 Expression Types . . . 75
5.6.1 Function Evaluation . . . 76
5.6.2 Operator Evaluation . . . 77
5.6.3 Domain Testing on Dimensioned Types . . . 77
5.7 Proof Obligations and Run Time Tests . . . 78
5.7.1 Run Time Testing of Constraints . . . 79
5.7.2 External Proof Obligations . . . 79

6 Type System Implementation **81**
6.1 Constraint Handling Rules . . . 81
6.1.1 Soundness and Termination Properties of CHR . . . 82
6.1.2 Representing HUSC Type Semantics as CHR Rules . . . 84
6.1.3 Representing HUSC Types in Prolog . . . 84
6.1.4 Pseudo-literate Prolog . . . 85
6.2 Implementation Overview . . . 86
6.2.1 Constraints . . . 86
6.2.2 Subtype, Join and Meet Predicates . . . 87
6.2.3 Attribute Predicates . . . 87
6.2.4 Context Predicates . . . 88
6.2.5 Summary . . . 89

6.3 CHR Type Inferencing 90
6.3.1 Unavailable Prolog options 91
6.3.2 Operator Definitions 92
6.4 Main Constraints 92
6.5 CHR Constraint Generation 94
6.5.1 HUSC Constraint Predicates 94
6.5.2 HUSC Factor predicates 95
6.5.3 Operator and Function Applications 96
6.5.4 Attributes Constraints 100
6.6 Constraint Resolution 101
6.6.1 External Proof Obligations on Input Variables 102
6.6.2 Satisfying Attribute Constraints 104
6.6.3 Resolving Test and Proof Obligations 106
6.6.4 Selecting Function / Operator Interfaces for Joins 107
6.6.5 Expression Type Judgements 109
6.6.6 Variable Type Judgements 110
6.7 Attribute Class Definitions Part I 111
6.7.1 Symmetric 111
6.7.2 Scalar Range 114
6.8 CHR Arithmetic 116
6.8.1 Boolean Algebra 117
6.9 Named and Generic Subtypes, Joins and Meets 123
6.9.1 Generic Join Relation 126
6.9.2 Generic Meet Relation 127
6.9.3 Joins / Meets over Internal Terms 128
6.9.4 Testing Types against their Definition 130
6.10 Attribute Class Part II 131
6.10.1 Subattribute List 131
6.10.2 Attribute Propagation Rule Map 133
6.10.3 Attribute Class : General Description 134
6.11 Context Management 135

7 Examples 143
7.1 Long and Short Integer Arithmetic 143
7.1.1 Abstract Model HUSC code 144
7.1.2 Abs Module Type Judgements 145
7.1.3 Concrete Model HUSC code 147
7.1.4 Conc Module Output 148

8 Conclusions and Future Work 152
8.1 Summary 152
8.2 Compiling HUSC Programs 154
8.3 Outstanding HUSC Implementation / Design Issues 157
8.3.1 Context Management and Symbolic Computation 157
8.3.2 Building the Code Hypergraph 157
8.3.3 Attribute Class Definitions 158
8.4 NonDeterminism, Inequality Constraints and Systems of Constraints 161

A Formal Operational Semantics 164
A.1 Operational Semantics 164
A.1.1 Definition, Declaration and Context Rules 167
A.1.2 Simple Expression Evaluation 168
A.1.3 Completeness Rules 170
A.2 Type System Semantics 171
A.2.1 Fundamental Types, Top and Bottom 172
A.2.2 Shapes 172
A.2.3 Attributes 174
A.2.4 Defining New Types 174
A.2.5 Subtypes, SubShape and SubAttribute Relations 174
A.2.6 Joins and Meets 175
A.2.7 Type Judgements 178
A.2.8 Definitions (except Functions) 179
A.2.9 Expressions 180
A.3 Proof Obligations 182

B Concrete Syntax eBNF 185
B.0.1 Lexical Structure 185
B.0.2 BNF Terminals / Tokens 187
B.0.3 Module Headers 188
B.0.4 Declarations 188
B.0.5 Module Imports 188
B.0.6 Data Definitions 189
B.0.7 Operator Definitions 189
B.0.8 Joins and Conditional Expressions 190
B.0.9 Operators in Expressions 190
B.0.10 Factors 190
B.0.11 Types and Type Definitions 191
B.0.12 Attribute Classes 191

C Implementation 192
C.1 HUSC Programs and Modules . . . 194
C.1.1 Data Structures . . . 195
C.1.2 Main HUSC Program . . . 196
C.2 HUSC Modules . . . 199
C.2.1 Parsing Modules . . . 201
C.2.2 Statements . . . 204
C.2.3 Module Syntax Checking . . . 206
C.2.4 Import Defintions (ImportDef) . . . 208
C.2.5 Module Importing and Identifier Lifting . . . 212
C.2.6 Displaying Modules . . . 213
C.3 Data Definitions and Declarations . . . 215
C.3.1 Data Declarations (DataDecl) . . . 217
C.3.2 Data Definitions . . . 219
C.3.3 Parsing Definitions . . . 222
C.3.4 Variable Renaming for Scoping . . . 226
C.3.5 Checking Defintions for Completeness . . . 230
C.3.6 Displaying Definitions . . . 231
C.4 Operator Definitions . . . 233
C.4.1 Parsing Operator Definitions . . . 235
C.4.2 Expressions . . . 237
C.4.3 Parsing Expressions . . . 240
C.4.4 Show Instances . . . 245
C.5 Types . . . 245
C.5.1 FullType structure . . . 245
C.5.2 Shapes . . . 247
C.5.3 Parsing Type Expressions . . . 250
C.5.4 Printing Types . . . 252
C.6 Type Definition . . . 254
C.6.1 Valid Type Definitions . . . 257
C.7 Attribute Classes . . . 259
C.7.1 Attribute Class Body (AttrClassBody) . . . 261
C.7.2 Displaying Attribute Classes . . . 262
C.8 Type System Encoding . . . 262
C.8.1 Encoding Modules . . . 264
C.8.2 Definition Encoding Overview . . . 265
C.8.3 New Type Definitions . . . 265
C.8.4 Dependent Variable Encoding . . . 266
C.8.5 Attributes . . . 275
C.8.6 Encoding Expressions . . . 278

Chapter 1

Introduction

There have been many advances in type theory in the last decade. Many of these advances have been associated with functional programming languages due the simplicity of their semantics and support for correctness proofs. For the computer scientist, this environment is rich, powerful and precise. For the physical scientist or engineer whose programming knowledge is largely based on third generation imperative languages, it is a daunting environment with a long learning curve. This impedes the adoption of advanced typing techniques in areas such as image processing, dynamic systems, and numerical simulation. The HUSC language is designed to bring the benefits of more strictly typed environments to this domain of programming as part of the Coconut project.

There are a number of serious challenges facing developers of complex scientific and engineering related software. The physical systems they model are complex with many relationships and constraints to consider. The models themselves are often quite complicated and have many subtle properties. There are many techniques and algorithms to implement the model, the choices depending heavily on the properties of the system. Finally, the underlying computer systems are expanding in capability and complexity, often incorporating multiple processors and nodes. Developing a system to run safely and efficiently with all of these considerations requires an effective structuring of the development environment.

The traditional methodology for development in these environments addresses each of these levels but in a largely informal way:

- define the physical relationships and produce a mathematical model, often in a LaTeXdocument;
- identify and prove properties of the system, recording these in a natural language document;
- develop an algorithm based on the model using standard library functions, algorithms from published sources, and developing new techniques, manually verifying that algorithms are appropriate;
- frame this algorithm in a program that transforms the available input data into the desired output data, and can be implemented on a target hardware platform.

At this final stage the program may contain a formal definition of some of the structure of the model in the type or class system, but the non-structural properties of the model will be described informally in natural language text at best.

What is missing is the ability to formally specify important model properties, and use these properties throughout the transformation of the model into an executable system. HUSC provides the first part of such a system by supplying an extensible type system that can track these properties independently of the implementation of the system, and use these properties to provide safe and optimized algorithm transformations from a library of definitions.

1.1 Document Structure

This thesis includes the following chapters:

Introduction provides an overview of the Coconut project as the framework for HUSC, a motivating example, a high level summary of the HUSC language and type system highlighting novel features of the language, and briefly describes the current state of the implementation.

Background Research presents observations on historical and existing development environments for scientific computing, and some relevant new work in type systems to explain some of motivations and the decisions made.

Semantics and Abstract Syntax provides an abstract syntax for the language and the term rewriting semantics for expression evaluation.

Concrete Syntax provides the formal lexicon and syntax grammar that forms the current parser, and an informal description of the syntax illustrated with examples.

Type System defines types, attributes, the subtyping system, attribute class definition, type inferencing and checking and the proof obligation system.

Type System Implementation presents the design and source code for the current type system implementation, with detailed explanations.

Example and Results presents a sample HUSC program with the type judgements, and general observations and conclusions.

Integrating HUSC and Future Work provides an initial set of requirements for the next level of the Coconut system, which will take the output of HUSC and produce a concrete program.

Appendix - Formal Semantics provides the formal specifcation of the operational semantics and type system.

Appendix - Parser Implementation the literate source code implementation of the language parser, syntax checks and translation of modules for the type system.

The type system is the most significant component of the project. This section includes a description of the types, the subtyping system including the definition of a subtype lattice, and the type inferencing semantics. This chapter can be read on its own, with some reasonable assumptions being made about the meaning of definitions and declarations in a declarative language.

The abstract syntax chapter and formal semantics appendix describe the language as a term rewriting system. Of particular interest in this section is the semantic for evaluating joined expressions for function and operator applications. The semantics are presented before the type system, in order to provide the context, but reversing

the order of reading might be preferable for readers comfortable with term rewriting semantics for declarative languages. The chapter on concrete syntax is aimed at future users of Coconut, and can be read or used as a reference for the examples section.

1.2 Coconut Project

The Coconut project is an endeavor at McMaster University led by Dr. Christopher Anand and involving a number of faculty and students. The requirements for the HUSC language and type system were established by the Coconut project, so an overview of Coconut is in required to explain this context.

The Coconut project aims to define a new development paradigm that

1. captures contributions from different specialists in a language suited to their individual needs,
2. tracks and verifies assumptions made in all stages of development, and
3. delivers high performance parallel code.

Coconut generates code from high-level specifications by allowing application developers to provide definitions at each of several stages of abstraction, independently controlling

1. mathematical relations (HUSC),
2. algorithm selection
3. code graph refactoring,
4. assembly instruction scheduling
5. process decomposition for multi-node networks

The four main stages of the Coconut system are:

1. the HUSC language,

2. codegraph transformations,

3. assembler instruction scheduling and

4. the run-time operating system CocoNet.

The structure of the Coconut system is summarized in Figure 1.2.

HUSC

A Coconut program consists of a set of high level specifications organized into modules (a), written in HUSC. The HUSC modules are parsed and typed to produce attributed syntax hypergraphs (ASGs) as an intermediate output data type. Processing of the HUSC source files identifies syntax errors, type errors, and unsatisfiable type constraints. The output is a hypergraph because HUSC allows choices, called joins, to define alternative implementations of operations and functions. Unverifiable restrictions required for the safe execution of the program, called external proof obligations, are documented as an addendum to the program file (d).

Codegraph Optimization

At the next stage, (h), the abstract syntax hypergraph must be rewritten to contain only implementable nodes and edges. This means representable data types (e.g. Int, IEEE float) and native system machine instructions. The graph transformations are based on a set of rewriting rules, including restructuring of abstract subgraphs and and transformations from abstract to concrete (system type/instruction) graphs. The hypergraph is also reduced to a graph at this level by selecting only one implementation for each operation. This decision process can be based on a number of different strategies, such as performance profiling or developer input, but any selection is acceptable because all of the implementations are acceptable in the context of the program. This factorization is similar to the identification of basic blocks by conventional optimizing compilers.

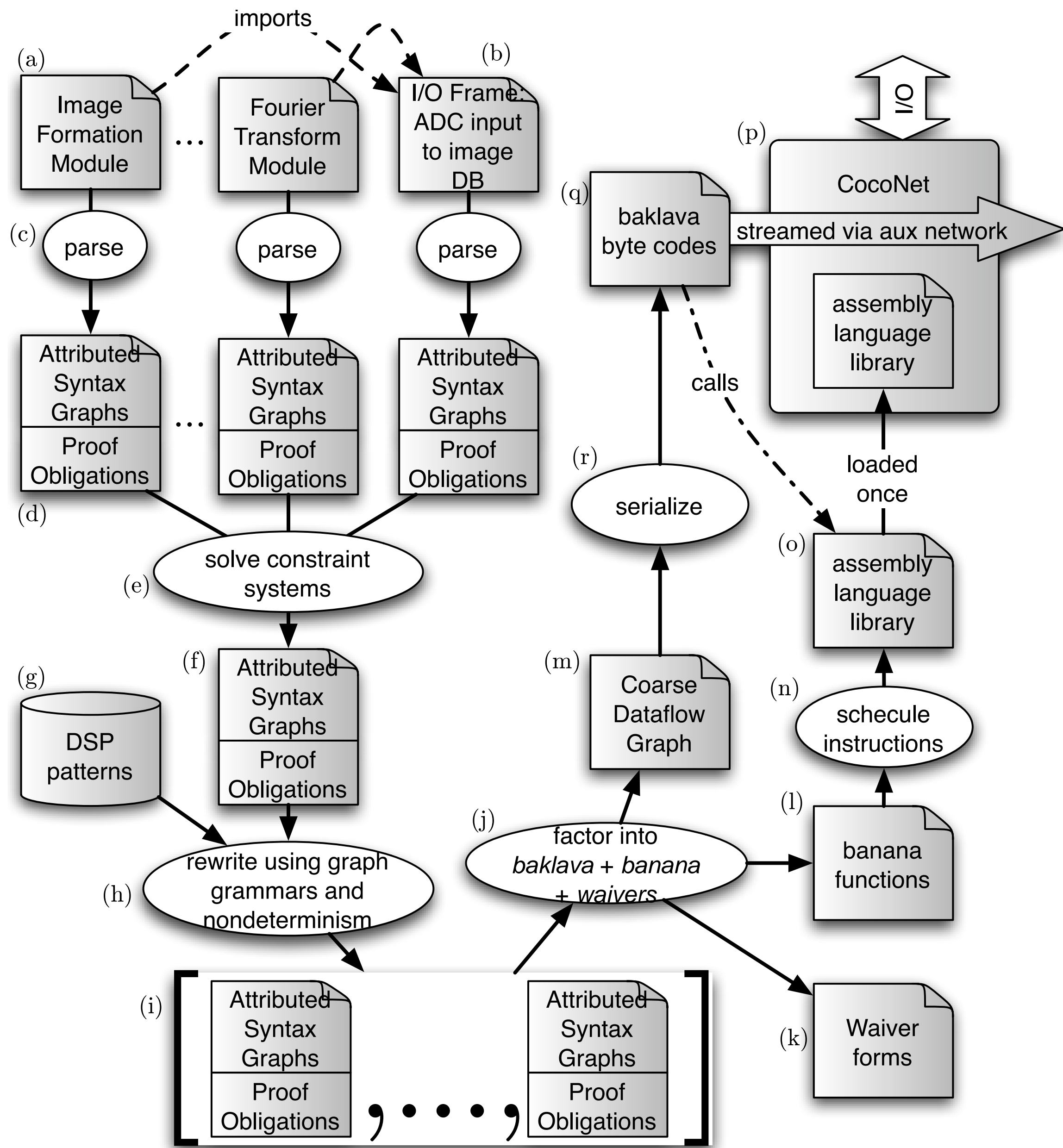

Figure 1.1: Illustration of data flow within Coconut compiler.

Assembler Instruction Scheduling

The assembly language functions are scheduled (n) into pipelined assembly code (o) in simple binary modules. The scheduler attempts to find the optimal assignment of registers and processors starting with an exhaustive search, and then using heuristics to prune paths that are unlikely to be successful. The modules are loaded onto the target computer cluster (p) before program execution.

At this stage the dataflow graph contains the scheduled assembly functions, together with a limited set of combinators representing concurrent dataflow (m). These graphs are serialized (r) into a stream of function calls, communication, synchronization and sequencing primitives to be streamed through the run-time system, CocoNet (p).

CocoNet

CocoNet is a run time operating system for dedicated multi-node networks, the generic target for Coconut programs. The language and run-time are designed to facilitate pre-run-time verification that the parallelization (which occurred in (h)), is correct, safe and independent of timing. All memory allocation is done at compile time (again in (h)), which allows pre-run-time verification of performance requirements and makes the run-time simple to verify by hand.

1.3 Example: LA Pack in HUSC

HUSC provides a mechanism for separating the fundamental modelling decisions from the implementation details through modules and function specialization. One practical use for this would be providing an interface into the LAPack library. LAPack, short for Linear Algebra package, is a set of highly optimized library routines for solving standard linear algebra problems, a cornerstone of scientific computing. The example provided here shows a real, although very simple, example of HUSC code and syntax that would support such a system. This is intended to explain the motivatation for the features implemented in HUSC.

One of the components of LA Pack are "driver" functions [1] for solving a system of linear equations $Ax = b$. There are a number of different solutions depending on the nature of the matrix A, the precision (IEEE single or double), and the accuracy desired, and solutions are provided for both real and complex systems.

To illustrate the complexity of the system, the properties of potential interest for the matrix are: bidiagonal, diagonal, general band, general tridiagonal, Hermetian, Hermetian band, Hessenberg, upper / lower triangular, orthogonal (packed, unpacked), symmetric (packed and unpacked), positive definite, triangular band (packed, unpacked), trapezoidal, complex unitary (packed, unpacked); and some combinations of these. There are real and complex forms, and single and double precision forms of each of these, leading to 20 choices of drivers. This is further divided into simple and expert drivers, with expert drivers computing additional information about the system to refine the solution and provide error bounds, leading to 40 choices for solving a system of linear equations. In addition to the above properties, in some implementations there are special "hand-written" versions for small systems, where the loops have been unrolled for further performance gains, or other system specific advantages have been exploited. This is a complicated environment because each of these properties has an important set of benefits associated with it, and ignoring those properties has a significant impact on the performance of the system.

Programmers using LAPack would typically determine in advance what the best implementation would be for their environment and fix that choice in the source code. There is no formal verification that the correct version has been used. In addition, low level changes, such as switching to use a packed model to store a tridiagonal matrix, will result in every function call needing to be changed, with very unpredictable results if one is missed. Given the complexity of the environment, and the fact that the function names are five or six nearly identical characters, this could be a significant source of error.

One good use for HUSC would be to define each of these functions as a specialization of a linear solver function, then let the type system use the properties of the matrix and element types to determine which implementation to use. In HUSC, these properties can be explicitly added to types and used when type checking the use of

[1] The LAPack term for functions that solve a complete problem

a function. Then decisions such the choice of precisions, or whether to pack the data (meaning to use a customized memory model based on an attribute) can be incorporated into higher, more context specific modules, without obscuring the fundamental system in lower level modules.

Consider the following three modules: MySystem, TheSystem and LAPackLinSolve. LAPackLinSolve provides all of the specializations of the solve function for linear systems of equations. This includes a subset of the real LA Pack drivers, with the appropriate labels, plus a few hypothetical extensions for small matrices. The solve function is used in TheSystem, the module that provides the fundamental model. This is then used in MySystem, which adds context specific details for the solution. The example does not include the type and attribute definitions to maintain clarity.

1.3.1 Literate HUSC Example

This is a typical literate husc module, with source code embedded in a Latex document (this thesis). The parser will ignore anything outside of the husc code environment delimiters. Literate programming is an excellent feature for documenting complex systems where a few lines of code might require pages of proofs or explanation.

The following specializations use the attributes Symmetric, Positive Definite, Packed and Precision(n), where precision is defined to be just single (1) or double (2) precision floating point. These attributes appear in braces beside the type they refine.

```
module LAPackLinSolve
  export (solve)
import (size::Integer)

Let solve be Function from (Matrix[n,n] of Real, Vector[n] of Real)
                                   to Vector[n] of Real;
n=size; -- rename for convenience
-- general case
solve(A::Matrix[n,n] of Real{Precision(1)},
         b::Vector[n] of Real{Precision(1)})
```

```
         = sgesv::Vector[n] of Real{Precision(1)};
-- symmetric
solve(A::Matrix[n,n]{Symmetric} of Real{Precision(1)},
         b::Vector[n] of Real{Precision(1)})
         = ssysv::Vector[n] of Real{Precision(1)};
solve(A::Matrix[n,n]{Symmetric} of Real{Precision(2)},
         b::Vector[n] of Real{Precision(2)})
         = dsysv::Vector[n] of Real{Precision(2)};
solve(A::Matrix[3,3]{Symmetric} of Real{Precision(1)},
         b::Vector[3] of Real{Precision(1)})
         = ssysv3x3::Vector[3] of Real{Precision(1)};
solve(A::Matrix[3,3]{Symmetric} of Real{Precision(2)},
         b::Vector[3] of Real{Precision(2)})
         = dsysv3x3::Vector[3] of Real{Precision(2)};
solve(A::Matrix[2,2]{Symmetric} of Real{Precision(1)},
         b::Vector[2] of Real{Precision(1)})
         = ssysv2x2::Vector[2] of Real{Precision(1)};
solve(A::Matrix[2,2]{Symmetric} of Real{Precision(2)},
         b::Vector[2] of Real{Precision(2)})
         = dsysv2x2::Vector[2] of Real{Precision(2)};
-- packed symmetric
solve(A::Matrix[n,n]{Symmetric,Packed} of Real{Precision(1)},
         b::Vector[n] of Real{Precision(1)})
         = sspsv::Vector[n] of Real{Precision(1)};
solve(A::Matrix[n,n]{Symmetric,Packed} of Real{Precision(2)},
         b::Vector[n] of Real{Precision(2)})
         = dspsv::Vector[n] of Real{Precision(2)};
solve(A::Matrix[3,3]{Symmetric,Packed} of Real{Precision(1)},
         b::Vector[3] of Real{Precision(1)})
         = sspsv3x3::Vector[3] of Real{Precision(1)};
solve(A::Matrix[3,3]{Symmetric,Packed} of Real{Precision(2)},
         b::Vector[3] of Real{Precision(2)})
         = dspsv3x3::Vector[3] of Real{Precision(2)};
solve(A::Matrix[2,2]{Symmetric,Packed} of Real{Precision(1)},
```

```
         b::Vector[2] of Real{Precision(1)})
         = sspsv2x2::Vector[2] of Real{Precision(1)};
solve(A::Matrix[2,2]{Symmetric,Packed} of Real{Precision(2)},
         b::Vector[2] of Real{Precision(2)})
         = dspsv2x2::Vector[2] of Real{Precision(2)};
-- positive definite and symmetric
solve(A::Matrix[n,n]{Symmetric,PosDef} of Real,
         b::Vector[n] of Real{Precision(1)})
         = sposv::Vector[n] of Real{Precision(1)};
solve(A::Matrix[n,n]{Symmetric,PosDef} of Real{Precision(2)},
         b::Vector[n] of Real{Precision(2)})
         = dposv::Vector[n] of Real{Precision(2)};
-- and so on...
```

The system of equations defined in the model below turns out to be symmetric, because it is the product of the transpose of a matrix by itself. The propagation rules for symmetric identify this characteristic, and resolve that A is a symmetric matrix. This attribute will allow the symmetric versions of solve to be used. At this point, the compilation cannot complete because the precision has not been defined.

```
module TheSystem
  export (x)
  import (B::Matrix[n,n] of Real,
             y::Vector[n] of Real)
{
A = (B^t)*B; -- for matrices ^t is the transpose operator
x = solve(A,y);
Use LAPackLinSolve for (solve) with {size = n};
}
```

The specific context for the final program specifies the matrix to be a rotation of a real 3d position vector. Providing specific values for both precision and the size of the matrix provides the final refinement necessary to pick specific implementations of the solve function. In this case, the single precision, symmetric packed and unpacked

will be selected, both the 3x3 cases and the nxn cases. The general single precision case would also be available.

The back end of the system now gets to decide which implementations to use, using heuristics or allowing the choice to be dictated by the programmer in some manner. In particular, the decision to take advantage of the symmetric storage model (Packed) or not is best handled at a detailed code graph level to examine the implications of this decision.

Note that type declarations are not necessary, the type of the output will be inferred from the context. In this case, it would have been a better documentation practice to include the explicit type declaration.

```
module MySystem
    export (answer)
{
Let rotate be Matrix[3,3] of Real{Precision(1)};
Let pos be Vector[3] of Real{Precision(1)};

answer = x;

Use TheSystem for (x) with {B = rotate, y = pos};
}
```

This example illustrates how HUSC modules can be used to isolate the complexity of the implementation from the models. In a more sophisticated model, there would be numerous operations and function calls, all of which could be specialized implicitly. In our example above, $A+C$ would select the specialization of matrix addition for packed symmetric matrices if A was implemented that way, without the model developer having thought of that. These implicit choice points for every operation and function call allow the models to be presented without implementation details, but also without excluding later specialization.

Another important feature of HUSC that would be useful in this example is the tracking of assertions. For example, while the matrix A in TheSystem is provably symmetric, it may also be positive definite for reasons that cannot be demonstrated in the code. This can be asserted using the syntax

```
A = B^t * B asserting {PosDef}
```

This attribute cannot be verified within the code, so (A,PosDef) is recorded as an external proof obligation for the module, and the attribute is accepted. This will then include all of the solve implementations requiring the matrix be positive definite, as well as all of the other choices that were available without it. It is then up to the model developer to provide an external proof that the matrix is positive definite, but HUSC will not perform any additional verification. This provides flexibility while explicitly identifying the assumptions made and tying them to the program output.

While these features don't create new capabilities for developers, they do greatly simplify and improve the safety of the code, and through this reduce the cost of optimizing the system.

1.4 HUSC — High level User Specification of Constraints

The HUSC programming language has been designed to address problems encountered by the developers of complicated numerical systems.

1.4.1 Domain specific declarative programming model

HUSC stands for High-level User Specification of Constraints, so named because the purpose of the language is to capture specifications in the form of relationships and constraints (and because it is the outside layer of the Coconut project). It is a purely declarative language, meaning identifiers are defined as they would in a mathematics or science paper, without a procedure for evaluating their definitions. HUSC defines an operational semantic and a type inferencing system, acting as the front end of a compiler for which different back ends may be supplied.

Any back end must support the operational semantic of HUSC, and provide "prelude" files that define the types, built in operations and an external function library (run-time library) in a HUSC compatible format. Coconut will be one of those back ends, when it has been completed, but in fact HUSC has no dependency on any of the other layers of Coconut.

This embedding of a front end in a back end system is necessary for logic and functional programming languages. These "declarative" languages specify relationships between identifiers without any specific processing or instructions. The language must be embedded in a back end structure that defines an actual activity. For example, a Haskell module may contain all of the necessary definitions to solve a problem, but it must be embedded in a monad (generally an IO monad) in order to get an output from the inputs. A Prolog module contains a static set of relationships and facts, but these are embedded in the WAM engine to make them a program. HUSC, as a declarative language, is similar in that it must be embedded in some environment that supports an interface with the operating system, and drives the output from the input using the definitions supplied in HUSC. As is the case with Haskell and Prolog, HUSC defines the operational semantic that must be supported by the back end without actually executing the semantic itself.

1.4.2 Join Expressions

A large part of developing efficient numerical programs is the use of specialized functions and operations based on the properties of the underlying model. For example, there may be hundreds of implementations to solve systems of linear equations, depending on the size and properties of the matrix, the desired accuracy of the answer and the type of the elements. Traditionally the programmer is forced to choose the strategy in advance, possibly before all the information is known, and certainly making rework necessary if the context changes.

In HUSC, specialized implementations may be defined for any function or operation. These specializations share the same operator symbol or function name, but their parameter and result types are subtypes of the general implementation. Then each operator and function application is implicitly considered the "join" of all of the definitions that satisfies the type context. A join expression represents a choice between "semantically equivalent" definitions, and the value of a join is the value of *any one* of the choices that satisfies a type context. This makes every operator or function application a prospective choice point, in which different implementations may be inserted.

Functions and operators may be implemented in HUSC or strictly as an interface

to be filled by an external call. The decision of which function or operator implementations from a joined expression are included in the final program is left to the back end system. Because there are choice points at every edge in the code hypergraph, a very sophisticated optimization scheme can be applied.

1.4.3 Shapes and Attributes

Types in HUSC are composed of a shape and an additional, optional set of properties called attributes. The shape is one of: scalar, tuple, record, array or function; and represents the underlying structure of the type. Attributes are properties or constraints that may apply to a specific type, have a property value and a set of rules that dictate to the type system how these attributes are "propagated" through expressions. New attribute classes may be defined within HUSC modules by providing a set of mandatory definitions for the class. This allows the developer to add new attributes, which can be embedded into the type system.

Symmetric, diagonal, tridiagonal, positive definite, are all examples of properties that can be exhibited by a square matrix (Matrix[n,n]). If two matrices are known to be symmetric, their sum is determined to be a symmetric matrix, This is a fundamental property, independent of the implementation of sum. Symmetric is an attribute in HUSC, and the sum rule is written explicitly as an attribute propagation rule. This rule can used by the type system to derive the property of symmetry in expressions whenever two matrices are added.

Another example of an attribute is the range of a scalar variable. The rules for the propagation of ranges are obvious and are also implemented in the HUSC type system. This is important because manipulating and storing a number depends heavily on being able to bound it within the natural system limits. Defining and tracking the range of numeric variables provides a guarantee against overflow errors, or the option of selecting alternate representations of large numbers.

1.4.4 Type Inferencing

HUSC incorporates a Hindley / Milner style type inferencing to reduce the workload of the programmer. This incorporates the variable type declarations, definitions and

the usage of the variables in expressions. It also includes the selection of function specializations and attribute propagation for any defined attribute classes.

The combination of join expressions with the type system provides a powerful mechanism for type safety and optimization. Attributes work together with joins to allow choices to be recorded without forcing design decisions to be made at inappropriate times.

One benefit of attributes is to enforce constraints in modules. Consider the following module that constructs an image from three layers of image:

```
module ImageGen
  export (image::ImageType)
  import (first, second, third::ImageType )
{
    Define ImageType as Array[-127..128, -127..128] of PixelType;
    Define PixelType as Integer{Range(0..255)};

    image = floor ((first + second + third) / 3);
    Use ImageDefs importing (ImageType::Type, PixelType::Type);
}
```

The element type of image is an integer between 0 and 255, but so are the element types of first, second and third. The range of their sum will be 0 to 765, and the type checking would fail at this point. Dividing the sum by 3 brings the range back, but then the result will be a rational type, not an integer. Taking the ceiling or floor of the expression allows the type checking to complete successfully. This is relatively simple constraint checking, but prevents a common error from occurring.

Another benefit of the attributes, combined with the joins, is in providing reliable choices of algorithms. For example, a discrete Fourier transformation (DFT) is a linear functional that operates on an equispaced grid (array) of data [2]. The property of "equispaced" could be associated with arrays of tuples as an attribute, and this would be a mandatory property for using the DFT function, rejecting any inputs that did not have that property. An explicit join expression could be constructed for an MRI system where

[2] in magnetic resonance image processing, this is often a two dimensional grid in frequency space and the inverse DFT is being applied to get a two dimensional spatial image.

EquispacedDataArray applies the DFT algorithm directly

DataArray applies a resampling function that converts an arbitrary sampling into an estimated sampling over a regular grid

The type system would discard the straightforward solution if the equispaced attribute was not declared for the grid, and both options would be available if it was equispaced, presumably resulting in the straight forward DFT application. Note that all of the specialized versions of the resampling function could also result in an implicit join matching the types of the data, presenting another choice or set of choices to the compiler.

Optimization is improved by establishing choices that can be made when more information is available. For example, symmetry of matrices provides an optimization opportunity, but for smaller matrices this might be outweighed by index testing. A module can be defined that specifies matrices of an as yet undecided dimension as symmetrical; operations on these matrices will include symmetric implementations in their joins. If that module is imported into a context where the matrix size is known, and compiled into a program, the decision to use the symmetric model or ignore that property can be made.

1.4.5 Assertions, Tests and External Proof Obligations

The type system tries to determine whether the attributes defined for an expression are valid properties or satisfied constraints given the context and the attribute class rules. There are four means of judging the attribute valid:

input requirement when the data item is an input to the module, the attribute is part of the type definition of the module interface;

propagation rules the definition of the attribute class provides rules that support the inclusion of the attribute in the expression type

run time test the expression is inside a guard that specifies a run time test must be included to prove the attribute is satisfied or an exception is generated [3];

[3]the failure of the guard currently results in “undefined”, a better strategy for exception handling should be included in future work

the attribute class may supply one or more such tests, or may supply none in which case such a test cannot be requested.

assertion any attribute appropriate to the type context may be asserted for any expression, meaning it is accepted as a valid attribute, but a proof obligation is then generated for the module.

The "proof obligation" states that the user of the module must guarantee, external to the program, that the asserted attribute holds on the specified data structure. The interface for a module then includes both the input specification and the additional external proof obligations. This tracking of attributes provides the flexibility to develop systems that cannot be formally verified within HUSC, while maintaining a detailed list of proofs that must be provided to reliably use the system.

Wherever possible, they type system will eliminate external proof obligations and run time tests. A run time test will be eliminated if the attribute rules dictate that the attribute holds. An assertion may result in constraints being propagated back to an input definition, causing those constraints to be added to the input specification and the original assertion to be removed from the proof obligations. Finally, when a module is imported into a new context, any additional attribute information is tested to see if the inputs provides satisfy the constraints so more tests or obligations can be eliminated.

In a general programming language such proof obligations might not be useful because the semantic meaning of the identifiers and constraints might be unclear. However with HUSC, we can rely on the context provided by the domain to ensure that the context provides a semantic meaning for the attribute classes, function names and operator symbols.

1.5 Implementation Overview

The main components of the implementation are a parser / syntax checker and the type system. The parser is written in Haskell using Parsec parser combinators, and generates a file of type definitions and encodes the program definitions as constraints for the type system. The type system is implemented in Prolog and the Constraint

Handling Rules (CHR) language, derives the “best” type description for each identifier, expression and subexpression in the modules, and also provides the list of function and operator interfaces that satisfy each operation or function application in the current context.

The output of the type system is written to three files:

1. final type judgements and accepted function / operator interfaces for each operator / function application

2. external proof obligations on identifiers in the module

3. a trace file containing rejected join candidates (operator and function interfaces that were determined to be unacceptable in the given context) to be used for debugging purposes

There are no proven, robust Haskell implementations of CHR at this point in time. The parsing was originally carried out in Haskell anticipating the use of a Haskell based CHR implementation, but this proved to be faulty under more complex situations. As a result, the SWI-Prolog implementation of CHR was selected, and has been highly satisfactory, despite the significant annoyance of having to move the data between the Haskell and Prolog environments.

The current implementation requires that the parser be executed first, followed by the type system as a separate program. This is inconvenient, but at this early stage in the development of HUSC seemed to be a acceptable.

The final stage of HUSC processing is to build the attributed syntax hypergraphs from the parsed definitions and type judgements. This has not been completed at this time, primarily due to time constraints, but also because there are no specific requirements from the Coconut project at this time to define the format of the hypergraphs. Work to complete this stage is currently underway.

1.6 Summary

HUSC provides a very powerful type system to support complex mathematical models. The addition of independent attribute classes and the ability to identify those

properties as requirements for function and operator specializations allows a substantially increased level of type safety. Making the function and operator applications implicitly select all specializations matching the type context increases the ability to optimize programs, using a much higher level of information that is generally available. The remainder of this thesis will provide more details about how this is implemented.

Chapter 2

Background Research

The goal of HUSC is to bridge the gap between scientific / engineering programmers and current advances in type theory and type systems. To support this goal, a short examination of some of the more popular existing (and historical) mathematical programming environments and some of the developments in type theory that were seen as valuable is provided here.

2.1 Mathematics Programming Languages

HUSC is intended for the development of systems with substantial mathematical, scientific or engineering content. While there have been a number of smaller scale projects in this domain, few have surfaced as a significant force in this area. A brief discussion of some of these, though certainly not a comprehensive overview, is presented here to support some of the decisions made in the project.

2.1.1 APL, J

Coconut / HUSC draws inspiration from, and has some parallels with, APL, the language proposed in the 1950's and described in 1962 by Kenneth Iverson [Ber79]. APL introduced matrix, vector and array based operations as primitives, and supported functions defined against those types. APL also maintained parsed expression trees for definitions which could then be resolved in later passes.

There were a number of decisions made in APL that reduced it's acceptability. There was no precedence of operations, so $2 * 3 + 5$ evaluated to 16, something that must have been very confusing. It also made use of numerous confusing symbols that required both an encyclopedic memory and special terminals / keyboards to manage. This made it difficult to learn and implement on a broad scale, and few people ever became acknowledged experts.

An interesting example of APL is provided below:

```
Primes: $(~R?R\int.{\diamond}R)/R? 1??R$
```

This produces a list of prime numbers below the input value. While the syntax is confusing, the deep semantic support that allows just over a dozen characters to deliver such a function is impressive.

J [Ive91] is the successor to APL, and was created and supported by Eric Iverson (son of Kenneth). It is essentially the same language, but with only ASCII characters and newer tool sets. It too shares many of the goals of Coconut including recognizing the notion of the precision of a floating point number, which in HUSC would be defined as an attribute (however, in J, precision is defined a module level and is constant for all expressions in the module.)

The software is available at www.jsoftware.com, and is more accessible now that no special hardware is required. Unfortunately, J replaces the confusing symbols of APL with utterly baffling ASCII sequences. For example, `1 o. i.7` evaluates to the sequence of values `sin(0), sin(1),...,sin(6)`.

The relative lack of acceptance of J may be due to it's complexity:

> J is a very rich language. You could study and use it for years, and still consider yourself a beginner. This is in sharp contrast to simpler languages like Basic or Java, where months of concerted study and use would make you an expert. *J* Primer Eric Iverson Copyright © 1991—2002 Jsoftware Inc.

APL and J both suffer from producing difficult to understand code and requiring constant revision to stay proficient. Of course, APL was designed when mainframe computer provided 256Kb of working memory and very limited disk space, so reducing the number of characters involved in the code was significant.

Two guiding principles for the concrete syntax of HUSC were derived from this experience:

- using recognizable symbols and function names is mandatory for wide acceptance, even at the expense of brevity
- it is necessary to standardize the fixity, associativity and precedence of operator symbols to match standard usage and provide the intuitive order of operations

2.1.2 MatLab

The Coconut system most closely resembles the MatLab [Mat84] integrated system in concept. MatLab provides a mathematical language, a growing set of modules (toolboxes) containing callable functions, and can produce executable code for a large variety of hardware and software platforms. There is strong support for matrix, vector, and array data types and operations, and it recognizes both precision and tolerance in variable values.

MatLab does not, however, provide strong type checking. For example, this is a quote from a beginner MatLab manual:

> A three-dimensional array might represent 3d physical data, say the temperature in a room ..., or it might represent a sequence of matrices A(k) or samples of a time-dependent matrix A(t)." Getting Started with Matlab ©1984—1988 The Math Works Inc.

MatLab is procedural in nature, supporting loops and conditionals. This makes it more flexible, perhaps easier to use, but also allows for coding errors and requires that the mathematics be coded, instead of just recorded. It also forces the user, presumably a scientist or mathematician, to take responsibility for ensuring the correct toolboxes are used, allowing for error in library selection. Consider the following call to a LU Decomposition solver:

```
[X, Flag] = bicg(A, b, 1e-6, 20, L, U);
```

Given the number of arguments, it is easy to make an error in their order, and there is no way for the system to ensure that A meets the required properties for the use of

this decomposition algorithm, it must be tested at run time or assumed to be correct. HUSC's system of constraints type and matrix attributes would allow the system to enforce all of the necessary attributes without runtime overhead. For example, a matrix can be asserted to be positive definite, so may be decomposed with Cholesky decomposition. The assertion then becomes part of the external proof obligations of the module.

This more casual approach to types is probably popular with the users, but limits the capability of MatLab to optimize the code. It also means that MatLab code cannot be theoretically verified.

MatLab was first released in 1984, when personal computers were running at approximately 4 MHz clock speed, 256 Kb memory, 5 Mb hard drive disks. The system has been extended dramatically since that time, but the underlying structure of the MatLab language has not changed. However, MatLab is probably currently the single most significant language / package in the development of mathematical, scientific and engineering applications.

2.1.3 Mercury

Mercury [3] is a general-purpose declarative language, loosely based on Prolog but with substantial extensions. It is not a mathematics language, but was included here because the module structure for HUSC was designed to follow Mercury's quite closely. Other than both being declarative languages, Mercury explicitly addresses the notion of deterministic, semi-deterministic and non-deterministic definitions, which was useful in the design of non-linear system solvers in HUSC.

2.2 Type Theory and Type Systems

The type system of HUSC is derived from two separate but related areas of active research. These are:

1. a type has a *shape* that can be statically analyzed
2. a type system can be best described as a constraint handling system

3. a type can contain a proof of some (simple) proposition

Some of the influences in these areas are examined below.

2.2.1 Constraint Based Type Systems

The idea that a type system is fundamentally a constraint handling system was introduced and explored by Martin Sulzmann and colleagues, beginning with the introduction of the HM(X) system, and continuing with many works expanding this concept. A quick survey of some of that work is essential here, since many of the ideas implemented in the system are derived from these works.

HM(X)

In [14], it is shown how standard Hindley / Milner systems may be defined purely in constraint form. The basic HM inferencing may be naturally extended with the introduction of additional constraints, where a sufficient condition for type soundness is that each simplification rule have a single head.

Type System Design with CHRs

In [11], the language called Constraint Handling Rules (CHR) is used to implement the constraint handling type system. Several examples of extensions are provided, including a simple means of testing for (S.I.) unit consistency in arithmetic expressions. This not only established constraint handling as a valid approach to designing type systems, it also established that CHR is a natural selection for implementing these systems.

HCHR and Chameleon

Two new languages were introduced to assist in the implementation of constraint based type systems. HCHR [4] is a Haskell implementation of the CHR language in which the constraint rules are typed using Haskell types, with the type checking and inferencing being carried out by the Haskell compiler. Chameleon is a Haskell extension which replaces the type class system in the Glasgow Haskell Compiler (ghc) with

an extensible, constraint based, implementation of the class system; additional work has been done showing how to implement functional dependencies using constraints. This project is based on the ideas in [12], and there are many more papers available at `www.comp.nus.edu.sg/~sulzmann/chameleon/index.html`

The initial plan for the HUSC type system was to implement it using HCHR. However, there was a significant theoretical limitation in that the Boolean guards (predicates) for rules in HCHR were limited to testing variable equality; the CHR system implemented in Prolog allowed arbitrary predicates to be used. HUSC required the implementation of a subtype predicate, which was easily handled in Prolog, so HCHR was rejected (after some efforts to work around this limitation). In addition to the theoretical difficulty, the implementation of HCHR seemed to be quite error prone for complex Haskell types (types other than numbers, characters or lists).

Chameleon has been used effectively in a number of situations, and may be capable of supporting a HUSC implementation. However, it is intended to replace Haskell's type classes, and hides the underlying CHR semantics by providing a new syntactic layer. Since the primary goal of this project was to explore and validate the use of constraint handling as a type system in this domain, and it is possible that the focus on the Haskell class system would impose limitations, this approach was rejected. It would be interesting to examine the use of Chameleon again to see if it could be used in place of the Prolog based type system.

2.2.2 Constraint Handling Rules

In [6], Thom Frühwirth introduces the Constraint Handling Rules (CHR) language. This is not a type system in itself, but has been adopted for use in the HUSC type system, so is described here.

CHR is a declarative language extension designed to write constraint solving systems, as a subsystem embedded in another language such as Prolog or Lisp. The paper provides numerous examples of constraint solvers written in CHR, some of which were incorporated into HUSC. Although the paper does not explicitly identify the use of CHR for type systems, it does say "A constraint solver can thus be seen as (an) inference system."

There are now several robust implementations of CHR in different variants of Pro-

log. The web-site http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/index.html provides a central repository of CHR related information, including providing a web-based interactive solver for experimenting with CHR solvers.

2.2.3 FISh — Static Shape Analysis

The notion of a type having a *shape* was explored by Barry Jay in the FISh language [9]. FISh is an Algol-like imperative language that incorporates static analysis of shapes, namely scalars, arrays and functions, with parameter, dimension and bounds checking. It uses a Hindley/Milner style unification system for type inferencing, including parametric polymorphism by quantification over array and "phrase-type" variables, where phrases are imperative code fragments, similar in spirit to quantifying over functions.

The static shape analysis, including both checking and optimization, is carried out during the compilation stage through term rewriting to a data and shape normal form. This is specified as a language called Turbot, an imperative language that supports functional concepts such as general recursion on commands and maps and folds across arrays.

This approach to shape analysis was adopted in HUSC, but with the addition of tuples and records as shapes, and with a different but similar set of array shape manipulation functions. HUSC does not include the "phrase type" variables, which are imperative code fragments, as these are not applicable given the declarative semantics. It is quite likely that the β-reduction strategies specified for FISh will be of use in the Coconut codegraph transformer layer, where the translation from abstract to machine representable operations will take place.

2.2.4 Types containing Proofs

A type system may additionally be considered to be a sort of proof system, based on the Curry-Howard correspondence. In such a type system, the type becomes a compound structure, containing a certificate of the satisfaction of propositions. These propositions may be mandatory or absorbed by an operation as part of the type checking. Attributes in HUSC are propositions of that nature, which can be

supported, required, propagated or absorbed by the type system.

Concurrently to the Coconut project (and HUSC development), two other projects are underway with similar goals.

Ωmega

Tim Sheard in [8] presents the Ωmega language, a Haskell like functional language where invariants in the program are enforced through the use of types. Three new features are added to the type system: Generalized Algebraic Data Types, an extensible kind system, and the generation, propagation and discharging of static propositions. The extensions allow backwards compatibility with the base functional programming language, while enhancing the power of the language.

Epigram

The Epigram project ([2]) is working on a dependently typed system for representing certificates of propositions within the type system in a rather novel programming environment that interactively develops programs as if they were proofs. While the style is quite different, they note the goals are similar to those of the Ωmega language, namely to use the addition of dependent variable types to provide the information required to supply certificates of program invariants in the types themselves.

2.2.5 Background Summary

There are many other areas of research that parallel some aspects of HUSC, such as the formal theorem provers, that have influenced it's development but haven't been discussed here.

While shapes, dependent types, types as constraints and encoding certificates of propositions within types are all areas of active research, the combination of these in a light weight environment is a unique project goal. The addition of the join expression further enhances the uniqueness of the HUSC language and type system. Finally, none of the other projects in this area are focussed on an application domain, and in particular do not even cover the most basic requirements of scientific and engineering

environments, Real and Complex numbers. By bringing these elements together, HUSC (and Coconut) is a unique endeavor.

Chapter 3

HUSC Language Definition: Syntax and Semantics

The definition of the language is given in three parts: a summary of the abstract syntax, a detailed formal and informal semantics, and then the concrete syntax as both an eBNF summary and a user-guide like description.

This is followed in the next chapter by the formal description of the type system, which is not described here except as required to explain the syntax of the definition statements.

3.1 Abstract Syntax

The abstract syntax is summarized here, using a standard eBNF format:

- *terminals*
- `non-terminals`
- [`A` | `B`] is a choice between A and B
- [`Optional`]?
- [`ZeroOrMore`]*
- [`OneOrMore`]+

with the following are terminal symbols: *Identifier*, *OpSymbol*, *ModuleId*, *TypeId*, *TypeVar*.

3.1.1 Modules

Modules have a name (identifier), a list of definitions that must be supplied (imports), a list of identifiers (or symbols) that are exported, and a body containing definitions.

The exports are currently just a list of identifiers or operator symbols, with the types being inferred, and optionally declared, within the module.

`Module`	→	[*Module*] *ModuleId* `ImportDecl` `ExportDecl` `ModuleBody`
`ImportDecl`	→	[`Declaration`]*
`ExportDecl`	→	[[`Identifier` | `OpSymbol`]]+
`ModuleBody`	→	[`Statements`]*
`Statements`	→	`DataDeclaration` | `DataDef` | `TypeDef`
		| `OperationDef` | `ModuleImport` | `AttrClassDef`

The export list does not provide an explicit type declaration because attribute classes, operators and type definitions are also exported, and there is no way to declare those. This is undesirable for the documentation of module interfaces, which should include type declarations for the exports, with some syntax for exporting type, operators and attribute classes.

3.1.2 Module Imports

Importing a module makes the `ImportDefinitions` list of identifiers available in the current module context. The identifiers (or operator symbols) must be exported by the imported module. If the imported module requires any definitions (it's import list), these must be satisfied in the `SuppliedDefinitions` list of data definitions.

`ModuleImport`	→	*ModuleId* `SuppliedDefs` `ImportedDefs`
`SuppliedDefs`	→	[`Definitions`]*
`ImportedDefs`	→	[*Identifiers*]+

3.1.3 Data Definitions and Declarations

Data definitions label a data expression with an identifier. There are three kinds of definitions, with slightly different syntax:

total a single definition completely defines the identifier;

partial defines a record or array (dimensioned shape) over a subset of its domain, with the subset defined by expressions over quantified variables; multiple partial definitions must partition the domain;

function defines a map from a list of parameters of declared type to a result value of declared type, with any number of function definitions providing alternative implementations

Data definitions have an optional local body for locally used data definitions and declarations.

`Definition`	→	*Identifier* [`FunctionDef` | `PartialDef`]?
		`JoinExpr` `Quantifier` `LocalBody`
`FuncDef`	→	[`ParmVar`]*
`PartialDef`	→	[[`PartialDimDef`]+ | [`PartialRecordDef`]+]
`PartialDimDef`	→	`IndexSpec`
`PartialRecordDef`	→	`FieldSpec`
`ParmVar`	→	[[*Identifier* | `Declaration`]]*
`IndexSpec`	→	[`IdentifierExpr`]+
`IdentField`	→	*Identifier* [`FunctionDef` | `PartialDef`]?
`LocalBody`	→	[[`Declaration` | `Definition`]]+
`Quantifier`	→	[[`QuantItemAll` | `QuantItemSome`]]+
`QuantItemAll`	→	`Identifier`
`QuantItemSome`	→	`Definition`
`Declaration`	→	[*Identifier*]+ `FullType`

3.1.4 Operator Definitions

Operator definitions are semantically the same as function definitions, and the abstract syntax is nearly identical except for the operator symbol instead of a function identifier. There may (will) be many operator interface definitions for each symbol, with each interface definition mapping declared type operands to a declared result type; an implementation is optionally provided (with an optional label for documentation). Operator interfaces also have attributes, similar to the attributes associated with a Function shape, as seen in type definitions.

The fixity and precedence of an operator symbol is defined in an operator symbol table. This is currently fixed for the language, but should be extended to allow new symbols to be defined. All definitions for an operator symbol must use this fixity and precedence. The table is presented in C.4.2.

`OperatorDefinition`	→	`OperatorInterface` `OperatorImplement`
`OperatorInterface`	→	*OperatorSymbol* [`Attributes`]* [`FullType`]+ `FullType`
`OperatorImplementation`	→	`OpImplLabel` `LocalBody`

3.1.5 Joined and Conditional Expressions

A join expression consists of one or more conditional expressions. The expressions in a join may be annotated with additional information for the compiler to use in the algorithm selection process (this has not been defined yet, is just a placeholder).

A conditional expression is a select statement consisting of expressions with boolean guards. The conditional expression may also have type assertions, which define or refine the type of the conditional expression, either by providing a type declaration or specifying additional attributes for the expression. Attribute declarations are either asserted, meaning they are accepted unconditionally but recorded as an external proof obligation, or request a run time test be inserted to verify the attribute constraint is satisfied (not all attribute classes support such tests).

Join and conditional expressions must be the main expression in a definition in this implementation. This is not a semantic restriction, as a variable in an expression may be defined as a join or conditional expression, but it makes the concrete syntax easier to write and display. This restriction could be removed with only esthetic implications.

```
JoinExpr        →  [ [ CondExprJoinAnnotation]]+  | CondExpr
JoinAnnotation  →  Label [JoinAttributes?]* not yet implemented
CondExpr        →  [ [AnnotatedExprBoolGuard]+  | AnnotatedExpr ]
BoolGuard       →  [BoolExpr  | Otherwise]
BoolExpr        →  Expr
AnnotatedExpr   →  [FullType  | [[ Assertions ]? [ RunTimeTests ]?]]
Assertions      →  AttributeList
RunTimeTests    →  AttributeList
```

3.1.6 Expressions

Expressions consist of factors and operators that act on them. Operators may be infix, prefix or postfix; each operator symbol has an associated associativity, fixity and precedence that is fixed throughout the program.

```
Expr       →  Factor
               | Expr InfixOp Expr
               | PrefixOp Expr
               | Expr PostfixOp

InfixOp    →  OpSymbol
PrefixOp   →  OpSymbol
PostfixOp  →  OpSymbol
```

3.1.7 Factors, SubExpressions

Nodes in the expression may be literals, variable expressions or expressions in parentheses (not conditionals or joins). The literals are the traditional IEEE integers and floats; constructors (operators or functions) are part of the preludes or other modules. Variable expressions are a variable identifier application, with optional expressions for indices, parameters or field names.

`Factor`	$\rightarrow$	`Literal` | `Variable` | `Expr`
`Variable`	$\rightarrow$	`Identifier` [`IndexExpr` | `ParmsExpr` | `FieldLabel`]?
`IndexExpr`	$\rightarrow$	[`Expr`]$^+$
`ParmsExpr`	$\rightarrow$	[`Expr`]*
`FieldLabel`	$\rightarrow$	`Variable`
`Literal`	$\rightarrow$	*LitStr* | *LitInt* | *LitReal* | *LitTuple* | *LitRecord*

3.1.8 Types

There are two kinds of types in HUSC: generic and named. A generic type is a shape, with expressions for it's dependent parameters, and a set of attributes. A named type is a labelled type expression, with its own dependent parameters, based on a generic type or another named type.

Type definitions assign type expressions to type identifiers, with optional dependent parameters. Type expressions (called FullType below) are types, with expressions provided for any dependent parameters belonging to the base type; the expressions can include the new type's dependent parameters as variables and other data terms or type variables defined elsewhere in the module. The new type may optionally include additional or increasingly refined attributes.

Each new type must be more constrained than the type it is based on, and is therefore at least a possible subtype of the original type. This means that the shape must be a subshape of the original (generally meaning identical), and that all of the original attributes must be present or further refined (constrained) in the new type. One generic type is a subtype of another generic type if it satisfies these criteria, generally meaning an identical shape where the dependent parameters are subtypes and the attributes are subattributes.

It is possible to declare the new type is not a subtype of the original, using the new type to break the subtyping hierarchy within the named types. If the new type is not a subtype of the type it is based on, it will be considered a subtype of the generic type at the root of the type hierarchy of it's base type. A named type is always a subtype of it's generic shape and attributes. By default, a newly defined type is considered a subtype of the type named in the type expression.

```
TypeDef            →  [TypeId]+ [Declaration]* SubTypeFlag FullType
SubTypeFlag        →  Boolean
FullType           →  [GenericType | NamedType ]
NamedType          →  TypeId [Expr]* FullType AttributeList
GenType            →  TypeId Shape AttributeList
Shape              →  [ Scalar
                        | Record [Declaration]+
                        | Tuple [FullType]+
                        | DimType [IntExpr]+ FullType
                        | FunctionType [FullType]* FullType]

Attribute          →  AttrClassId [ AttributeProperty ]?
AttributeProperty  →  Expr
```

3.1.9 Attribute Classes

this has not been implemented in the parser, but is described because the hard-coded attributes match this description.

The attribute class definition creates a new kind of attribute. The class has an identifier, the type to which it is associated, a property of declared type, and a set of associated functions and rules.

The following functions are required in the attribute class definition. For an explanation of subattributes, joins and meets, see 5.2.

sub*AttrClass* true if the first argument, as a property of this attribute class, represents a subattribute compared to the second argument.

join*AttrClass*, **meet***AttrClass* function providing the attribute property value of the join or meet of the attributes represented by the two argument property values respectively.

test*AttrClass* optional predicate that tests a data value at run time to determine if it satisfies an attribute as described by the property value given.

The attribute rules define how the attribute properties of expressions may be calculated from attribute values of the components of the expression. This is a pattern matching syntax, where the rules are of the form AttrClassId(`Expr`(A,B,...)) = `Expr`(AttrClassId(A), AttrClassId(B), ...)

Finally, there may be local data definitions and declarations, as in the local bodies.

`AttributeClass`	→	[*Persistant*]? *AttrClassId* `FullType FullType AttrClassBody`
`AttrClassBody`	→	`IsAttrTest SubAttrTest JoinAttr MeetAttr`
		[[`Declaration` | `Definition`]]*
`IsAttrTest`	→	`Definition`
`SubAttrTest`	→	`Definition`
`JoinAttr`	→	`Definition`
`MeetAttr`	→	`Definition`
`AttrPropagation`	→	`Definition`

3.2 Semantics

The operational semantics of HUSC are a simple non-recursive term rewriting system. These semantic rules explain how the back end compiler, in particular the rest of Coconut, will interpret the code hypergraph representations of the HUSC programs.

Data definitions describe the type of data terms and their relationship to other data terms in the form of expressions, for example `f = m * a`. Variables and functions are defined as regular (non-recursive, non-cyclic) mathematical expressions with unary and binary operators. The definitions are static, and do not imply any particular activity on the part of the compiler. In particular, if a defined identifier is never used in a program, its definition need never be evaluated. The decision of how to use these relationships to accomplish the goals of the program is left to the back end.

Arrays may have multiple "partial" definitions for different subsets of their indices, but these must not overlap. For example,

```
for all i A[i,i] = 1.0;
for all i, j = [2 .. i-1] A[i,j] = 1.0 - 0.1*j + 0.1*i;
```

provides a well defined lower triangular or symmetric matrix, but is an incomplete general matrix (more partial definitions must be added). The use of quantified vari-

ables (i,j), when combined with fold and map operations, eliminates the need for loops when processing arrays.

Definitions must not be cyclic and recursion is not allowed directly. For example, the definitions of f, c and a are all invalid in the code below:

```
f(i::Integer) = fibbad::Integer where  {
   fibbad = select {
       (f(i-1) + f(i-2)) | i > 1
                           1 | 0 <= i <= 1
       } }
 a = b * c;  c = b / a;
```

because they are all recursive or cyclic. An iterative algorithm may be described as an array with elements based on other elements, A definition must be available to resolve the recursion

```
A[i] = A[i-1] * A[i-2] for i = 2..n;
A[0]=1.0; A[1] = 0.99;
```

the array of iterants must be defined over a finite (but arbitrarily large) number of iterations (n).

This simple structure is designed to make a guarantee of termination easier to provide by both the type system and the back end compiler, and was felt to be appropriate for the target domain. Extending HUSC to allow recursive definitions would be a significant but important step to consider for the future.

The only unusual requirement is the recognition that multiple choices are available for each function and operator evaluation. A function or operator implementation may be written in HUSC, or just the interface may be provided. If an interface is selected in a definition as part of a join expression, the back end would be required to link to an external library to obtain the implementation or use one of the other implementations from the join. While making optimal use of these choices is very difficult, and an interesting area of research in itself, there is no significant problem because the back end may select any of the choices available.

The remainder of this chapter is a detailed informal description of the semantics. The formal algebraic description is in the appendix A.1.

3.2.1 Programs

A HUSC program is a set of output data definitions, a set of input data definitions and a set of relationships that describes how to evaluate the output with respect to the inputs, literals and imported values. The program consists of a main module, whose exports are considered outputs, and imports area considered inputs, which must be translated to / from native system data types by the back end. The main module may import definitions from other modules, using their exports and supplying any import definitions they require.

The HUSC modules also supply type definitions to interpret the declared data types as known shapes and attributes. The type system will check and infer the types of every data identifier, expression and subexpression based on these definitions. It will also select all of the specialized function / operator implementations and interfaces that satisfy the type context for each function / operator application.

The output of the HUSC processing is a set of fully typed code hypergraphs with data expressions at the nodes and function / operator interfaces as the edges. There may be multiple edges between the nodes, representing choices available for a function or operation application. The roots of the graph are the exported identifiers of the main module, and the edges are imported identifiers or literals.

3.3 Modules

A module is a set of definitions, a subset of which are available for export, and a list of imported identifier declarations that must be supplied by any importing module. The module definitions are based on the imported definitions, literals, and other definitions imported from other modules (as opposed to the importing module). All definitions in a module must be *complete* meaning all of the symbols in the definition must have a meaning in the module context, for the module to be compiled successfully.

3.3.1 Module Bodies

The body of the module is a collection of:

- declarations

- definitions
- type definitions
- operator definitions
- attribute class definitions
- module imports

These statements can be in any order within the module. Each updates the context for the module or frame they are in.

3.3.2 Module Context

The context Γ includes the identifier definitions and type judgements for a module. It also includes attribute class, function and operator definitions. An expression E has a local context Γ_E that extends the module context with local variable declarations and definitions. All type, attribute, function and operator definitions must be at the module level.

A module context is formed of the output context, input context and the private context. The input context is formed of definitions supplied by the importing module when the module is imported. The output context is a subset of the definitions in the module context, with the remainder being private (only available within the module) or local (available to a single definition). There is no distinction between input, output and private contexts for evaluation and typing purposes.

Local contexts are distinguished from the global context through variable renaming, where each locally defined variable is renamed to a distinguished [1] name in the module context.

Complete Definitions

In Coconut terminology, a program is called *concrete* if it can be compiled into an executable program on a specified system. HUSC programs do not have to be concrete, as they may be defined with abstract types and function / operator interfaces

[1] The implementation currently does not guarantee uniqueness at this point.

without HUSC implementations. Instead, a HUSC program is called *complete* when each output definition is complete and well-typed. The definition of complete is the intuitive one for a declarative language, namely that all of the identifiers and symbols used have a meaningful definition in the program context. Well-typed means that the type system is able to produce a meaningful, unambiguous HUSC type judgement for every identifier and expression in the program.

Explicitly, this means each each output definition satisfies the following criteria:

- all of the identifiers in all of the definitions must be *complete*
- each function applied in an expression must have a declaration; each operator in an expression must be recognized operator symbol; either must be applied to the appropriate number of parameters / operands
- each data definition must be well-typed (HUSC types)
- each function or operator application must have at least one defined, well-typed interface that satisfies the type context

These conditions are necessary and sufficient for the construction of the output code hypergraphs.

Definitions must be *complete* within a module's context for the HUSC processing to be successful, otherwise an error will be logged for the identifier.

The definition of *complete* is given by the following recursion:

1. a literal value
2. an imported definition of a complete type
3. a recognized operator symbol acting on the appropriate number of complete operands
4. a function application of a declared function name with the appropriate number of arguments, each of which is a complete expression
5. parameters / operands with complete type declarations are complete within the context of the function / operator interface

6. quantified variables are complete in the context of the partial definition statement
7. locally defined variables are complete if they are complete in the local context
8. the definition of a function or operator interface, where each parameter / operand and the result is declared with a complete type; if the implementation is provided, it is complete if the definition is complete within the local context.
9. each defined type based on a standard HUSC shape, with all dependent expressions being complete and all attributes complete
10. a named type based on a complete type, optionally with complete attributes and dependent parameters, and each dependent parameter declared of a complete type
11. attributes are complete if the attribute class definition is complete, and the property value expression is complete.
12. an attribute class definition is *complete* if is defined for a ground target type, has a ground property value and provides a subtype, join and meet functions

The formal rules for this are given in A.1.3.

3.4 Declarations

A declaration constrains the type of a data identifier to be a subtype of the declared type. It does not explicitly fix the type as in other languages. The final type judgement will be constrained by any type declarations, but will be derived from the identifier's definition. There can be multiple declarations for an identifier, in which case the type judgement is a subtype of all of them, and undefined if they are incompatible. Declarations are not required for every data identifier, but are required for function definitions, operator definitions and imported definitions.

Function declarations are a special case as they declare the identifier to be a class of functions. This class declares the number of parameters and the base parameter

and result types for this function name. All definitions of this function must have the same number of parameters, and the defined parameter and result types must be subtypes of the declared types. Note that it is not necessary to have any definition for a function that satisfies the entire range of its declared types.

It is unusual that the parameter types must be subtypes of the declared parameters types, because then the definition will not be applicable in all instances that the function declaration will allow. This is fundamental to HUSC function and operator definitions — the type system selects only the definitions that are applicable in the type context of the application, allowing specialized versions of a function to be provided for optimization.

3.5 Definitions

A definition statement defines the value of an identifier when it is evaluated as an expression. All identifiers must have a *complete* definition, which can be one of the following three forms:

- a function definition with well-typed parameters,
- a set of partial definitions based on the shape of the expression
- a single total definition.

3.5.1 Function Definitions

A function definition specifies a new function interface for a declared function name. The interface consists of typed parameters and a result type, both of which must be subtypes of those specified in the class. The definition may also include an implementation of the interface, as a *complete* expression, generally using the local body to define the results. The implementation is optional; if no implementation is provided, it is assumed that the implementation will be made available through lower levels of the Coconut system (basically an external link). Parameter variables are considered defined in the local scope; their definition will be that of the arguments of a function application.

The value of a function applied to a set of arguments is *any* of the defined function interfaces that satisfies the type context. The decision as to which of the acceptable interfaces is made by lower levels of the Coconut compiler. At the HUSC level, it is only important to recognize that each interface must represent a semantically identical implementation within the allowable type context.

3.5.2 Partial Definition

For some shapes, it is sensible to provide more than one definition statement to simplify the definition. For example, each field of a record can be defined with a separate statement. A list might be defined iteratively as

```
Let f be List[0..n];
f[0] = 1; f[1] = 1;
for i = (2..n) f[i] = f[i-1] + f[i-2];
```

The nature of these shapes allows a domain analysis to conclude that the collection of provided definitions cover the entire domain of the identifier without overlapping

Within a module, the collection of all partial definitions for a variable must completely define the domain of that variable (although paradoxically, "undefined" is a valid expression for a definition). In addition, the domain subsets for any two partial definitions must not intersect. In other words, the partial definitions must partition the domain of the complete definition. These conditions are checked as part of the type checking, and failure to achieve them will result in a compilation error [2].

3.5.3 Total Definitions

A total definition provides a complete definition as a single expression for the identifier. All scalar definitions are total definitions (e.g. $x = 3.5$), but other shapes (except functions) can also have a total definition. For example, an array can be defined as the sum of two other arrays ($A = B + C$) without any specification of

[2]Testing that a domain is completely defined is fairly limited at this point. The above examples would be solved, but more complicated partitions would require more advanced domain analysis. This has been included as part of the future work.

indices or quantified variables. There can only be one total definition statement for an identifier, and if there is there can be no other definition statement.

3.6 Operator Definitions

An operator definition is similar to a function definition, but applies to a prefix, postfix or binary infix operator symbol. It defines the types of the operands and result, called the interface, and may include an implementation for the interface.

3.6.1 Operator Symbol Table

In order for the expression grammar to be decidable, any operator symbol must have a constant precedence, fixity and associativity, regardless of the types of the operands. Currently the symbol table is fixed, but a syntax for adding new symbols would be straight forward.

3.7 Importing Modules

A module may import definitions from other modules. The importing module must supply the definitions required by the imported module. If definition are imported, the imported identifiers must not also be defined in other definition statements (declarations are acceptable).

Any kind of definition may be imported. If a function name or operator symbol is included in the import list, all of the function/operator interfaces and implementations are imported.

Definitions from a module may be qualified, in which case all of its exported symbols are renamed, prefixed with the module name. Operator symbols may not be qualified.

3.8 Expressions

Expressions can be evaluated to produce a data value.

3.8.1 Simple Expressions

Simple expressions are the standard mathematical expressions composed of factors (literals, variables and expressions in parentheses) unary / binary operations and function applications. Operators and functions are intrinsically overloaded by the types of their operands / parameters, although each operator symbol must have a constant fixity, precedence and associativity. Were this not the case, the types of expressions could be ambiguous or undecidable where the types of the operands were unknown.

Variables

A variable expression is an identifier and an optional extension:

- a list of function arguments in parentheses,
- a list of integer expressions that fall in the domain of a dimensioned data identifier
- a single field name from a record definition

An unextended variable may be replaced by its total definition, or its collection of partial definitions (assuming they are Θ). The value of an indexed variable application is the projection of the total definition at the value of the index. The value of a field reference is determined based on the definition of the field.

The value of a function application is the value of the join expression formed by applying all of the function definitions that satisfy the type context to the argument expressions (see below).

Note that if an identifier is imported into the main module, it will have no definition and therefore cannot be evaluated further. The expectation is that this definition will be provided later, either by an importing module (reducing the current main module to an imported module) or the definition will become a system input in an executable program.

Literals

Literals will include integers, floating point (scientific notation), Booleans and strings, or tuples of the above. Other literal types will be built using constructor operators (or functions), described within the preludes.

Literals have an indisputable type judgement, which for numbers includes a range attribute specifying the value (3 : Integer{Range(3..3)} for example).

Conditional Expressions

A conditional expression is an ordered list of expressions with boolean guards, similar to a Haskell case statement. A boolean guard is just an expression that evaluates to a boolean value, in other words a condition or predicate. The value of the conditional expression is that of the first expression with a satisfied boolean guard. The evaluation of any the guards may take place at run time, but it may be possible with type attributes to reject some branches statically, based on the attributes inferred for variables in the boolean guards.

3.8.2 Joined Expressions

One or more expressions may be "joined" where each of the expressions has the "same semantic meaning" within the module within a given type context. The semantic of this relationship "same semantic meaning" is formally defined by the join semantics, but informally it is an equivalence class of members whose values are equally valid when the expression's guard is satisfied by the type context.

In the explicit join construction, each "candidate" expression has a guard, a list of attribute assertions and run time tests. Asserted attributes must be proven true for the join candidate to be accepted, tested attributes will be either proven, statically rejected, or tested for at run time. The expression candidate is "successful" when all of the assertions and tests result in True.

Function and operator applications are implicit joins, with each function / operator definition for that name / symbol being a candidate for the join. When a function interface (consisting of parameter and result types) satisfies the type context, it is called a "successful candidate" in the join. The type context consists of the

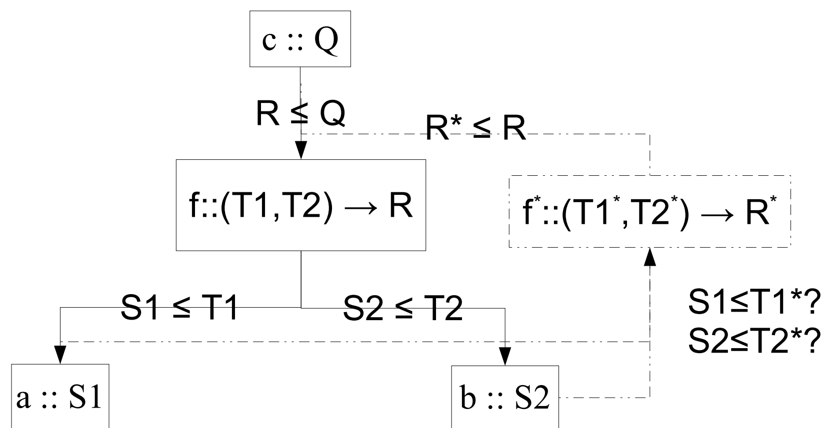

Figure 3.1: Illustration of the relationships between a function, function interface, arguments and result types.

type constraints on the result and type judgements of the arguments. Note that the implementation of the interface is **not** required, the promise of the implementation is made by the interface, and the implementation can be provided later. The value of the joined expression is the value of any one of the expressions. The lower levels of Coconut will choose which of the successful join candidates to include in the final executable program.

In order for a function application to be well typed, the result type of the function must be compatible with the expected result type, and the arguments passed to the function must be compatible with the parameters. Specifically, the result of the function must be a subtype of the expected result, and the argument types must be subtypes of the parameter types. This matches with the traditional definition of a function in a type system. This relationship is illustrated in Figure 3.8.2, by the solid lined boxes and arrows.

In HUSC, the declaration of a function represents its most general form, and its most general type interface. An arbitrary number of different interfaces for the function may be provided, where each interface represents a specialization of the general function. Intuitively, a specialized implementation of a function handles some special case of the general function and takes advantage of specific properties to perform an optimization. Formally in the HUSC type system, this means that the result type must be a subtype of the general result type, and the parameter types are subtypes of the general parameter types. This subtyping restricts the context in which the specialization is applicable, because knowing the arguments satisfy the general function parameters does not imply that they will satisfy the specialized version's parameters. The HUSC type system performs this check, selecting the specialized function interfaces that are satisfied by the argument types. The function interface is represented in the diagram with dashed lines. Note that the result type of the specialized function is guaranteed to be a subtype of the expected result, so this is not an issue.

3.8.3 Local Body and Local Context

The local body of an expression provides locally scoped data declarations and data definitions. Quantifier variables in partial definitions and function parameters are

implicitly declared and defined in the local scope; they may not be redefined within the local body.

Chapter 4

Concrete Syntax

This chapter provides a detailed informal description of the concrete syntax of the HUSC language with examples. The eBNF diagram formally defining the syntax is given in Appendix B.

4.1 Literate HUSC

One of the original requirements for HUSC was that it be a literate language, meaning that HUSC code could be embedded within a LaTeX source file and be compiled from the LaTeX file. This has been accomplished, and HUSC source is either prefixed line by line by the symbol "coco>>" or embedded in a "cococode" environment. This is currently identical to the "code" environment for other languages, and is treated as a literal section by LaTeX processors.

The new environment was intended to allow an extension to the HUSC lexicon by including traditional LaTeX math symbols. In particular, the Greek, Russian and miscellaneous symbol alphabets should all be available as letters and symbols within any programming language intent on supporting scientific and mathematical computing.

It would also be good to introduce a layout notation that could be used to replace braces and separators, similar to that of Haskell.

4.2 Informal Description and Examples

This section presents examples of the concrete syntax to illuminate the concepts, but also to act as an user's guide for the language.

4.2.1 Declarations

There are two basic syntax forms for declarations. The full line declaration is in the form *Let* identifier list *be* type. The compact form, used in parameter lists, records and anywhere where space is an issue, is in the more traditional identifiers *::* type format. The compact syntax can be used anywhere, the full (a.k.a. Baroque) form only as a line in a body.

```
module MeasurementError
     export (covar::Matrix[n,n] of Real)
     import (data::(x::Array[n] of Real,
                          y::Array[n] of Real), scale :: Fraction)
{
Let a,b be Real;
invScale, otherFrac :: Fraction;

invScale = 1 / scale;
xdata = data.x; -- implicit array definition
a = sqrt( sum(xdata) );
b = ...
covar = invScale (x y^{t}); }
```

4.2.2 Module Import Interface

A module is imported to obtain access to its exported definitions. The importing module supplies the inputs required by the imported module using the *with* list. The imported definitions may be used by name, or may be renamed on import. The definition names may optionally be qualified with the name of the imported module, or a pseudonym for the imported module.

The example below gets an error covariance matrix from a set of measurements, using the module MeasurementError. Note that the array dimensions are only passed implicitly.

```
module GammaSpectroscopy ... {
...
Use MeasurementError  for (covar)
    with {data.y = gammaReads,
             data.x = timeIntervals };
...
result = ... covar ...
...
}

module MeasurementError
     export (covar::Matrix[n,n] of Real)
     import (data::(x::Array[n] of Real,
                          y::Array[n] of Real))
{
covar = ... data.x ... data.y ...
}
```

4.2.3 Definitions and Expressions

In the example below, D is a total definition which will result in an array type, A has four partial definition statements and an explicit declaration again for an array type, and sin has a function declaration and three implementations.

```
...
import(n::Natural,
          B,C :: Array[1..n,1..n] of Real);

Let A be Array[1..n,1..n] of Real;
D = B + scale*C;
```

```
for all i A[i,0] = 0.0 ;
for i=[1..n-1] A[i,i] = 1.0;
for all i /= j, j=[1..n-1]
    A[i,j] = i * j ;
for i = n, j=n A[i,j] = 0.0;

% supertype for all sin function implementations
Let sin be Function from Real to Real{ Range(-1.0..1.0)};

sin(x:: Real Range([-pi()/4.0..pi()/4.0])
    = sinlookup::Real{Range(-1.0..1.0)} where
      { sinlookup = ....x .... };

sin(x:: Real Range([-pi()..pi]) = basicsin::Real{Range(-1.0..1.0)};

sin(x::Real) = allsin::Real{Range(-1.0..1.0)}
where {    allsin = .... };
```

Note that type, operator and attribute class definitions may not be made in a local body, but only at the global level.

4.2.4 Joined Expressions

The *join* represents an explicit join of expressions, where no annotations nor guards have been included. A semi-colon separated list of (possibly conditional) expressions follows.

```
Let foo be Function from Real to Real;

foo = join [fooA, fooB, fooC] where
{  fooA(x::Integer) =  (x + 1)::Integer;
    fooB(x::Integer) =
        (x + Lim[maxCnt])::Integer where {
               for i = [1..maxCnt] Lim[i] =  Lim[i-1]+ 2^(-i) }
    fooC(x::Integer) =  sillyx::Integer where {sillyx = x / 2 * 2 + 1;}
}
```

4.2.5 Conditional Expressions

The *select* represents a conditional expression. The comma separated list that follows consists of annotated expression, boolean guard pairs separated by a | symbol. The boolean guard is just an expression but it will be constrained to return a Boolean in the type inferencing. The conditional expressions are listed in the order they are to be tested, and the first guard to return TRUE is selected.

The annotated expression is a standard expression with an optional type declaration. The type declaration consists of either "::" and a full type declaration, or a list of asserted or tested attributes ("asserting {*attribute list*}", "testing {*attribute list*}). All function definition statements require the result expression have a full type annotation, like absx in the example below. The guarded expressions in the select statement are examples of asserted attributes; these assertions should be proven by the type inferencing, so not result in an external proof obligation.

The *otherwise* symbol is the equivalent of TRUE.

```
% Integer is a type annotation for k
abs(x::Real{Range(lb..ub)}) = absx::Real{Range(0..max(lb,ub))}
where {
   k = select {
            -x asserting Range(0..(-lb))   |  x < 0.0;
             x asserting Range(0..ub)       |  otherwise;
                         % x >= 0 a better alternative
         }
  }
```

4.2.6 Types

The type definition statements map a type to a type identifier, with optional dependent type parameters. The type may be based on another defined type, optionally with additional attributes defined, or a generic shape with optional attributes (a generic type).

The generic types have syntax associated with them, and this syntax is also applicable to types defined on those shapes. For example, any type based on a dimensioned

shape can have an index expression in square brackets immediately after the variable ($A[i, j+1]$).

The dependent type parameters are similar to function parameters for a type definition. The dependent parameters may be type variables `a <: Scalar` or may be data variables of a defined type. In the current implementation, only integers, reals, booleans and pairs of those types are supported as dependent variable types, because the type system must be able to manipulate these values.

The attributes presented below (other than range and symmetric) cannot be defined from HUSC source files with the current parser. However they are implementable in the type system, and have been included to show an example of how the language might be used.

```
Define UnitReal as Real{ Range(-1.0..1.0) };
Define NormalizedReal as UnitReal {Precision(0.5e-3)};
% further constrain range, will take intersection
Define PosNormReal as NormalizedReal {Range(0.0..1.0)}

% attribute strictly monotonic (increasing, f) means f(A[i-1]) < f(A[i]),
%and first takes the first element of a tuple
Define DataTable as
  {StrictlyMonotonic(Increasing,first)} Array[1..n] of (x::Real, y::NormalizedReal);

% below, x must be compatible, y is of a dependent type variable a,
% a TimedDataTable is a variant of the Data table,
% so perforce is incompatible (this was optional)
Define TimedDataTable(a <: Top)
    based on DataTable of (x::Real{ SIUnit(seconds) }, y::a);

Define TriSymUnitMatrix(n, \alpha<:Real{Range([-1.0..1.0])}) as
   { Symmetric, Tridiagonal} Matrix[n,n]  of \alpha;

Define MRImage(dim) as Array[dim,dim] of (r,g,b::Integer{ Range(0..255)});
Define NormMRImage as MRImage(-127..128,-127..128);
% NormMRImage is a subtype of MRImage when dim = -127..128
```

4.2.7 Attributes

Attributes are listed immediately after the base type in a comma separated list. Each attribute is an attribute class and an instance of its property type, if any, which is in parentheses like a function argument list. Examples of attributes and attribute lists are seen above.

4.2.8 Attribute Class Definition

Attribute Class definitions have a similar format as modules. The attribute class are defined for a type, and have a property associated with them (simple attributes have a boolean property of being True or False). The attribute class is therefore a relation between the type they are defined on and the property type.

The HUSC syntax for defining attribute classes has been eliminated from the current implementation. Part of the future work for HUSC is to identify kinds of attribute classes, and then allow new attribute classes to inherit that structure.

4.2.9 Operator Definition

Operator definitions include an interface and optionally an implementation for that interface. The interface consists of the symbol the operator is associated with, an optional list of comma-separated attributes, the types of one or more operands (potentially including attribute lists) and finally the result type of the operand. The operator interface is written as:
Operator op{attributes} (varA::typeA) op (varB::typeB) = (varResult::typeAopB)
where the position of the operands and operators is explicitly presented. Postfix, prefix and infix binary operators can be defined this way, but the operator definition must match the fixity of the operator symbol, as defined in the operator symbol table.

Note that the attributes associated with the operator are distinct from the attributes associated with the operands and result of the operation. In the example below, addition is noted to be commutative.

If provided, the operator implementation is given as a local body to the interface, providing a set of declarations and definitions that satisfy the interface. The definition

of the result variable defined in the interface must be provided in the implementation body, using the operand variables as arguments as required.

Because any given interface may have multiple implementations defined, each implementation is be given a label to distinguish between them. This label has no uniqueness requirements, but ideally should be sufficient to uniquely identify the implementation. If no implementation is provided, the operator is assumed to be abstract for the purposes of the module, which allows type checking and inferencing but does not on its own support execution (it is assumed another module will provide the implementation).

```
% Sparse attribute is show here with a property of a function rtrv
% that takes a two dimensional index and returns the value; it's a variable name.
Operator +{Commutative} (SymA::{Symmetric}Array[n,n]  of a <: Top)
                                    + (SparseA::{Sparse(rtrv)}Array[n,n]  of b <: Top)
                                    = (SumA::Array[n,n] of (c <: Top))
    where (``GJUSymmetricSparseRTVRPlus'') {
         SumA[i,j] = select {
                      SymA[i,j]                | rtrv(i,j) == undefined
                      SymA[i,j] + rtrv(i,j) | otherwise
                   }
}

% this is a pure interface definition, the implementation
% must be provided from an external source later
Operator (B::Array[n,n] Symmetric of Real) + (C::Array[n,n] Sparse of Real)
      = (SumA::Array[n,n] of Real Range(MaxDoubleRange())) ;
```

Chapter 5

HUSC Type System

The type system is the core of the HUSC language. This chapter provides an informal description of types, subtyping relationships and type judgements. The formal definition is provided in A.2.

5.1 Types

This section contains the informal definition of the HUSC type system. The formal rules are provided in A.2.

Types have three components: a name, a "generic" shape, and a list of additional constraints called attributes. Shapes are parametric classes defining the structure of a data definition, similar to types in traditional type systems. Attributes are additional parametric constraints that apply to a data item. Attribute classes are defined for a specific type and its subtypes, and represent a predicate of some sort.

A shape with an attribute list forms a "generic" type. A type definition statement defines a new "named" type, which is an optionally parametric type (constructor) based on another named or generic type.

5.1.1 Shapes

Shapes represent a fundamental structure of a data item. There are five classes of shapes that are part of the language definition: Scalars, Tuples, Records, DimShapes

(dimensioned shapes) and Functions. These are the only shapes the type system can "reason about", so the list of shapes is not extensible within HUSC modules. Shapes are equal when they are from the same shape class and all of their parameters are equal. Note that dimensioned types are only equal when their bounds align exactly, not just when the size of the domains are equal.

Lower stages of the Coconut system must be able to interpret and use the shapes in the system. They can be used to selecting the storage (memory, file) model for the data, and also during code generation and algorithm selection. Abstracting the shape within the type allows semantic differences in the data to be differentiated by the type checker, while allowing code and algorithm reuse once type checking is complete.

Scalars

A scalar is an atomic data item. Numbers (Integers, Reals) are the most important scalar, but strings and Booleans are also scalars. Only numbers are given any real attention in this thesis, because of the target domain,

Arrays / DimShapes

An array is a dimensioned shape, a special class of function that maps a $domain \subset \mathcal{Z}^n$ to a single result type, where the domain can be expressed as the cross product of integer ranges (i.e. an n-dimensional rectangular prism). Precisely,

$$\begin{gathered} Array(dom, \alpha) \subset Function :: dom \rightarrow \alpha \text{ where} \\ \exists l_{1..n}, u_{1..n} \in \mathcal{Z}.dom = (l_1..u_1) \times \ell \times (l_n..u_n) \text{ and} \\ range : l_i..u_i \triangleq \forall l_i \leq k \leq u_i.k \end{gathered}$$

The term DimShape was used instead of Array to avoid the connotations associated with Arrays. Lists, iterative definitions, and arrays can all be types of this shape. This shape is very important both because it is possible to statically analyze the domain and because it forms such a large part of the structure of scientific programs.

Function

A function is a mapping from a list of typed parameters to a single result type.

Tuple

A Tuple type is the standard ordered product type of unnamed, typed elements. The order of the elements of a tuple is the only means of accessing the elements.

Record

A Record type is a set (with no order) of named, typed fields. The lack of order of the fields dictates that the projection and injection of field values requires the field names, so "default" ordering may be provided. This restriction is in place because the subtyping order for records, although not yet in place, will include any records sharing the same field names, with new field names allowed in an extension, as in [7]. This kind of extensible record subtyping would be broken by any arbitrary sequencing of the field names.

Partial definitions for records may be given, with a subset of the fields being defined in an expression. This could be a single field at a time, an arbitrary number of fields may be defined using other records as the basis, assuming a subtype relationship can be inferred to exist with the result record.

```
image.contrast = 5.5;
image.dims = (256,256);

colourImage = image;
colourImage.rgb = (128,128,128);
```

The inferred type of colour image would have fields rgb, contrast and dims.

Note that syntactically field names are not types or typed data, but are parsed as identifiers.

Missing Shapes

Some key shapes that have not been included are:

- algebraic data types,
- recursive types.

There is no specific reason for their exclusion, other than to reduce the scope of the project. Algebraic data types in particular might be a valuable addition for the application domain.

Abstract Types

HUSC types are inherently abstract, with no associated guarantees that they are representable on a real system. Generally, the goal of the HUSC programmer is to refine the abstract types with enough concrete information that they can be converted into system representable types for inclusion in the final program, but is not necessary in all cases. For example, a module that is intended for import by other modules might not have sufficient bounds on a scalar variable to guarantee it is finite in any context, but imports sufficient information from an importing module to tighten these constraints.

Top and Bottom Types

There are two specific abstract types that are outside of the regular type system.

Top($\top_T$) is the type of anything, the ultimate supertype. It represents an ambiguous type judgement, insufficient information being available to even decide a shape[1]. While a program cannot be considered *complete* with any $\top_T$ judgements, it is an acceptable outcome for a module intended to be imported into other modules that will further refine the ambiguous types.

Bottom($\bot_T$) is the type with no elements, the ultimate subtype. It represents an unsatisfiable type judgement, due to contradictory constraints. Unlike top, there are no circumstances in which bottom is a desirable outcome, nor new context that will recover a meaningful type.

This approach to $\top_T$ and $\bot_T$ is drawn from [10], pg 191—193. They introduce a number of complications to the theory of the type system, but these are mostly details that do not adversely affect the implementation nor the practical interpretation of the type judgements.

[1] It is interesting to consider what attributes might be applied in this case.

Shape Abstract Types

Each shape also has an associated abstract type, the supertype of all types with that shape. These cannot be defined from within the HUSC syntax, but are built into the type system, such as Scalar, the supertype of all scalar types. These types are often used as a supertype for dependent type variables, e.g. $\alpha \preceq Scalar$ implies α can be any scalar type.

General Abstract Types

Beyond the notational conveniences of the above abstract types, there may be uses for types that are inherently abstract, that are intended to be factored out of any expressions they appear in. The presence of such a type would serve as an indicator to flag invalid use of a module or operation. For example, consider the type of matrices that have a basis. We could subtype these matrices to those having a specific concrete basis type, and then the basis would be just a dependent data term. However, another approach is to deal with the basis as supporting unimplemented equality and transformation relations only. Then operations on the matrices would check the equality of the matrices to determine if they are identical, for example

```
 Define Basis as TypeTop;
 Define {Reflexive}== as (b1::Basis) == (b2::Basis) = t::Boolean;

 Define {Reflexive}== as
     (A::Matrix(b1::Basis)[m,n] of a) + (B::Matrix(b2::Basis)[m,n] of b) = t::Boolean
     where { select
                    for all i, all j fold(.and., (A[i,j] == B[i,j])) | b1 == b2;
                    undefined      }

Let b1, b2 be Basis;
Let A,B be Matrix(b1)[3,3] of Real; Let C be Matrix(b2)[3,3] of Real;

AB = A + B; AC = A+C;
```

The equality between bases cannot be implemented, because a basis is a subtype of type Top, which cannot be instantiated. However, the reflexive attribute of the equality operator allows the condition $b1 == b2$ to be replaced with $True$ unconditionally in the expression $A + B$, but not in $A + C$, the latter therefore being unimplementable. The implementation of addition is therefore undefined unless the bases are syntactically identical.

In a traditional system, the basis instance would have to be labelled with a unique code, and the unique code tested at run-time. This use of attributes on operators allows a significant increase in the expressiveness of the type system.

5.1.2 Attributes

Attributes are additional, independent, properties or constraints associated with a type. There is no order with the set of attribute associated with a type.

When an attribute is declared as part of the type of an data identifier, it is a constraint on the definition of that identifier. The type system will either:

- derive evidence that this constraint is satisfied,
- conclude that proof of this must be left as an obligation to the user
- conclude this is not possible in this context, resulting in type judgement of $\perp_T$

Attributes may also be derived for the type of an expression from attribute propagation rules defined in the attribute class, based on the use of the identifier in the module context. If a type judgement includes an attribute as part of a type, either:

- there will be a corresponding external proof obligation demanding support for the attribute
- run-time test that supports the attribute,
- sufficient support has been found from other supported attributes based on the propagation rules of the class.

5.1.3 Named Types and the Subtype Hierarchy

Shapes and attributes may be used as ”generic” types, but new types may be created from these and other named types. This may be for a number of different reasons:

- to label a complex generic type description so it doesn't need to be repeated;
- to associate abstract semantic meaning from the ”real world” context with the type;
- to create a labelled refined version of the type, called a subtype;
- to create a type variant that is considered completely new at the semantic level, but has a common structure with another type that can be exploited by later stages of compilation.

The new type is by default considered a subtype of the original, but the definition can sever this relationship and create a new, incompatible variant. Variants share the underlying structure of the base type, with the same shape and attributes, information that can be exploited at lower levels of the Coconut system while allowing HUSC to consider the types distinct.

The newly defined type will inherently be a subtype of the original type. The new type may add new attributes from new classes, or may add additional constraints from classes already in the base type; in the latter case, the type is actually assigned the meet of the attributes of the same classes, thereby guaranteeing the new type is a subtype of the base. This named subtype relationship is explicitly recorded in the type system and any context in which the base type is accepted, the named subtype will be accepted.

Variants

If a new variant is desired instead, the type definition explicitly erases the subtype relationship (in the above rule, the subtype? flag is a Boolean value with True being a subtype, False a variant). The new type will be entirely incompatible at the HUSC level, and will required new definitions for all functions and operators (although there could be short cuts to simplify this process).

Variants allow the modeler to define types that are semantically different, but have an underlying structure that is the same. For example, a two dimensional array is conceptually a different thing than a matrix, which has a basis in some space. A matrix is can be represented as a two dimensional array, but defined as a variant. Although the type checker will distinguish between the two types, the compiler will use the same storage models and will perform additions using the same algorithms..

Dependent Types and Cycle Detection

Dimensioned shapes are parameterized in their domain, and function, record and tuple shapes are also parameterized in their component types. This shape parameterization is tightly controlled, with the shapes effectively being type constructors. Named types may be also be parameterized with data or type variables, which can be used as shape parameters, but can also be used as attribute property values or even extracted to the data level.

Because these dependent types are not constrained, it is clearly possible to define a type that cannot be instantiated. Both the dependent parameters and attributes included in type definitions include type references themselves, possibly building cycles into the type definitions. Testing for cycles in type definitions has been incorporated into the syntax checking, with a type definition not being Θ if there is a cycle in types of the dependent variables or attributes. This checking occurs before the type system is invoked, so the type system assumes no such problems will be encountered.

This checking prevents recursively defined types. This decision was made to simplify the type system, which is already quite complicated, but without any particular problem having been discovered. The semantics for this are included in the rules on *complete*, 3.3.2.

5.2 Subtyping

One of the core ideas behind HUSC is the notion of subtyping. While subtyping with dependent variables is generally undecidable, a number of restrictions have been introduced to make the algorithm decidable. While this system has not been proven

sound and complete, this project demonstrates that it is worth exploring this approach further and that there are practical applications of the current approach.

The main relationship is the partial order subtype, denoted $\preceq$, with strict subtype denoted $\prec$, over the set of types. The subtype relation is reflexive, transitive and antisymmetric; strict subtype is asymmetric and transitive.

HUSC partitions subtype judgements into three domains:

1. the subshape relationship denoted $\preceq^{Sh}$
2. the subattribute relationship $\preceq^{A}$ for attribute class A
3. explicitly defined named subtype hierarchy

All three of the partial orderings (subshape, subattribute and subtype) are reflexive, transitive and anti-symmetric.

Consider two types S and T, and the evaluation of the subtype relation between them $S \preceq T$. For $S \preceq T$ to be true, the shape of S must be a subshape of the shape of T, each attribute of T must have a matching subattribute in S (although S may have additional attributes of classes not represented in T). If S is a generic type, the above conditions are sufficient. If S is a named type, there must also be an explicit subtype relationship between S and T.

Note that for named types, the subshape and subattribute assertions are validated when the type definition is validated, at which point it is sufficient to maintain the explicit hierarchy as a tree.

5.2.1 SubShape

The shape of S must be a "subshape" of the shape of T. Shape relationships are fairly simple: the shape class must be the same, any type parameters must be subtypes, and any data parameters for the type must be equal.

- T is a Scalar iff S is a Scalar
- T is a DimType(dom, α_T) iff S is a DimType (dom, α_S) and $\alpha_S \preceq \alpha_T$; note the domains must be identical, not just the same size.

- T is a Function iff S is a Function, and the results and each of the parameters of S are subtypes of their match in T. Note that a subtype of a function specialization is covariant in both the result and parameter types, as opposed to the standard contravariant parameters, requiring the type system to verify the context before selecting the subfunction.
- T is a Tuple / Record iff S is a Tuple/Record and the element / field types of S are subtypes of their match in T [2]

The flexibility to define subshapes is deliberately limited to what is necessary to support function subtyping. This allows the shapes to be resolved relatively easily, mostly with a few CHR predicates that deal with subtypes.

5.2.2 Attribute Subtyping

The subtype relationship over attributes ($\preceq^A$) is only defined over two attributes of the same class. Informally, the relation $A(\mu) \preceq^A A(\rho)$ means that the constraint described by $A(\mu)$ is more restrictive than that of $A(\rho)$, or that any term satisfying $A(\mu)$ also satisfies $A(\rho)$. For example, the subattribute relation for the range of a scalar variable is equivalent to "is contained in".

Each attribute must supply a definition of $\preceq^A$ that, given two values of the class' property type, results in a Boolean value, True if the first is a subattribute of the second. This was discussed further in 5.4. The ability to define such a subtype relationship is a key restriction on the definition of attribute classes.

5.3 Joins and Meets

The subtype relationship is a partial ordering on the set Types, so $< Types, \preceq >$ forms a poset, a mathematical structure with many useful properties. This can be used to define joins $\sqcup$ and meets $\sqcap$ on the set Types, creating a lattice, another useful mathematical structure. A lattice has two binary operations, a join and a meet, which represent the supremum and infimum of any pair of elements of the set.

[2] Extensible records should be introduced later to allow additional fields in S, but this has not been implemented

Both operations are commutative, idempotent, and associative. A useful synopsis sufficient for understanding this document is available online at [1], with a deeper analysis presented in [5]. The value of the lattice theory was not explored as part of this work but the notation was useful and it simplified the semantics of the attribute class definitions.

The join of two types is their least general generalization (LGG), here called "minimal supertype"; it is a subtype of any other supertype of the two and has no subtypes that are also supertypes of both operand types. In HUSC, the type of a join expression is the join of the component expressions.

The meet is the most general unifier (MGU), here called the maximal subtype; it is a subtype of both types, and there is no other subtype of both that is not also a subtype of the meet.

Meets are used to handle multiple constraints on an identifier from multiple uses in a module. Joins are used to determine the result type of a joined expression, providing the most restrictive type judgement guaranteed to contain the result.

As in the subtyping, the join and meet operators are defined in terms of shape, attribute, and named subtype decisions.

- the shape of the join /meet of two types is the join of its shapes
- the attributes join / meet are the join / meet of all attributes of the same class, where unmatched attributes result in erasure / equality respectively
- for named types, the join is the minimal supertype of both, and undefined (ambiguous) if there is none
- for named types, one of the two must be a subtype of the other, and this is the value of the meet, and undefined

5.3.1 Joins and Meets on Shapes

The join of two shapes is only well defined if they are the same shape, any type parameters are joinable, and any data parameters are equal. If this is not the case, the join is ambiguous. Similarly, the meet of two shapes is only well defined if they are the same shape, any type parameters have a valid (non-$\bot_T$) meet, and any data

parameters are equal. If this is not the case, the meet is empty. This strict relationship is very useful for inferring dependent type parameter values.

5.3.2 Joins and Meets on Attributes

Attribute classes are always operated on independently of each other. Every attribute class must have a definition of a join and meet function, which takes two attribute property values and returns another property value which represents the join or meet. Note that this operation takes place independently of the value of the data term the attribute is applied to, a serious restriction but one that allows for decidability.

5.3.3 Joins and Meets for Named Types

The named subtype hierarchy forms a tree, where the subtree of any node represents its named subtypes. For every named type, there is an ancestor that is a "generic" type, an unnamed shape with a set of attributes; this can been seen by considering how types are defined.

The join of a named type will be an ancestor in its subtype hierarchy tree. If the other type is named, it will be a common ancestor that may be a named type or may be a common generic shape, but either with fewer and less restrictive attributes.

For a meet of named types to be defined (i.e. not $\perp_T$), one of the types must be a pure subtype of the other, and that is the value of the meet. The meet is so limited because the subtype hierarchy is a tree, so while joins can meet at a common root, meets must share a branch.

Proof has not been offered for the axiom stating that the join and meet exist. The restricted structure of the type system supports such a claim, but the proof of soundness and completeness were beyond the scope of this thesis.

5.4 Attribute Classes

Attribute class definitions are dependent on the module context to resolve the meaning of the types and function definitions, and the typing judgements of the context are dependent on the attribute class definitions in the context.

The attribute class definition components are denoted:

A the name of the attribute class

ρ the type of the property value

β the target type of the class (the type the attributes are applied to)

$\preceq^A$ an implementation of the subattribute relation for this class

$\sqcup^A$ an implementation of the join operation for this class

$\sqcap^A$ an implementation of the meet operation for this class

$\models^A$ an implementation of a test to support the attribute at run-time for this class

rules A set of pattern matching rules for deriving or inferring attribute properties for terms in a simple expression, based on the functions / operations in the expressions.

A an implicitly defined function is created to return the value of the attribute property as a data term

Note that each attribute class then uses five elements of the function name space.

5.4.1 Attribute Context

The attribute context $\mathcal{A}$ is part of the module context, and contains the attribute class definitions and rules. Attribute classes are associated with a type called the target type, and only have meaning when applied to this type or a subtype Each attribute class has an associated property type, a value of the property type defines a specific constraint of this class.

5.4.2 Attribute Class Interpretation

Attribute classes may be defined in modules, assuming the structure presented for attribute class definition can be satisfied. These extensions to the type system are implemented by generating new CHR statements that define the subtype relationship

for the attribute (subattribute) and inference rules for the attribute class. In effect this is "modifying the code" of the type system, but if the CHR code follows the specifications provided here, the new attributes will be managed appropriately, and this simulates the specification defined here.

The propagation rules are only practical because the HUSC language is intended for use in a domain specific context. In particular, operations like + have an accepted meaning, so an attribute rule that works over addition can be defined. While misusing standard mathematical symbols won't break the type safety of HUSC, having consistent definitions will make the language much more useful.

The function *Attribute* is overloaded in the two separate contexts. The expression $Attribute(\rho)$ in an attribute context $\mathcal{A}$ specifically as an attribute declaration, assertion or test, represents the satisfaction of the constraint by the expression. In the standard expression context Γ $Attribute(\mathrm{E})::\ \beta \rightarrow \rho$ represents a function that returns the property value of *Attribute* that expression E satisfies, or undefined if this does not exist. If the value of this second function is undefined, an error will be generated during compilation. The overloading of the names may seem confusing, but the context of the two uses in the language is intuitive.

Note on Current Status of Attribute Classes

The implementation of this component of the system was not completed, meaning generating these CHRs from HUSC syntax because of two unresolved issues:

1. how to best specify the logic that dictates the inference rules for an attribute class and how to convert those rules to the CHR semantic;

2. it is clear that there are *kinds* of attribute classes, which correspond to (at least) the kinds of CHR constraint solvers, that can be implemented as part of HUSC, but classifying these is a substantial research project in itself.

It did not make sense to proceed with defining a HUSC declarative syntax that would support the generation of attribute code, knowing that this effort would be entirely subsumed by future work.

5.5 Type Judgements

Type inferencing in HUSC is a two phase approach where constraints are applied, and derivations are tested against the constraints, until the most general unifier (minimal supertype) is found. Type constraints are imposed by the (optional) declaration and usage of identifiers in constrained expressions. A type judgement is formed by deriving the type of an identifier from its definition, using the constraints to restrict the expression's type. If insufficient information has been provided in the form of explicit type declaration this will be ambiguous ($\top_T$) and if the constraints are incompatible, the type will be undefined ($\bot_T$). This eliminates the distinction between type inferencing and type checking, as all type judgements are inferred, with declarations being just one source of constraints.

Unlike many type systems, HUSC seeks the most constrained type (in particular attributes) that still contains all possible solutions given the relationships defined in the module, the least general generalization. This allows more specialized function and operator implementations, allows more efficient storage models and code generation by the next levels of Coconut.

5.5.1 Overview

The type inferencing system generally works as follows:

1. identifying constraints on data definitions from declarations and assertions

2. constraints are propagated into expressions and subexpressions based on the available function and operator interfaces

3. type judgements are rendered for imported variables and literals, then propagated back up through the subexpressions as the operands / arguments are judged

4. as type information becomes available, attribute propagation rules are fired over operator symbols and function class names, without regard to operator / function interfaces

5. the result types of operator and function applications are made based on the join of the individual interface types, where the interfaces must satisfy the constraints from the declarations / assertions but also the propagated attributes

The type and attribute judgements are included in the context Γ or the local context Γ_E of an expression for variables defined in the body of an expression.

5.5.2 Declarations and Definitions

A type declaration in HUSC represents a supertype constraint on the actual type of an identifier. The actual type judgment must also take into account the derived type from the definition of the identifier, inferred constraints from its use in the module, and attribute propagation rules.

Functions and operators have a special set of rules that supports the implicit joins around their applications in an expression, and the ability to define multiple interfaces and implementations around a single function name or operator symbol. The validity of a function definition (or operator definition) is the same as for the definitions of other shapes.

Declaring an attribute causes an external proof obligation to be added to the context if, and only if, the attribute cannot be derived from the definition.

5.5.3 Function Declaration and Definitions

Function names in HUSC are actually abstract classes of functions, with support for multiple interfaces and implementations. The class declaration establishes the most general type declaration of the function. Function implementations may then be developed to work for either the entire range of the type interface or only a subset of it. The function definitions then act as an interface, with the parameter and result types specifying the context the implementation may be used in.

Unlike other identifiers, functions must be declared. Declaring a function $f : (T_{1..n}) \rightarrow R$ defines a class of functions represented by the function name f. Each member of the class maps subtypes of the parameter types $T_{1..n}$, to a subtype of the result type R. Note that both the class members are covariant in both parameter and result type, an unusual situation which is resolved by the function selection process.

Declaring a function establishes a function context that will resolve the overloading of the symbol f based on its parameter and result types. Note that all function declarations and definitions must occur at the module level, not in the local body of an expression.

Defining a function establishes an interface for that function, and optionally a new implementation of that function. The interface is a pair $(T'_{1..n}, R')$ representing subtypes of the parameter and result types respectively. Where an expression contains an evaluation of the function class, all of the interfaces that are satisfied by the current type context will be considered. A function interface is satisfied by the context when the type judgements for all of the arguments are subtypes of the matching parameters, and the function interface return type satisfies any constraints on the result expression. Any definition of a functions must be explicit, unlike other shapes which can be inferred or derived from a situation.

Functions may have attributes, like other shapes, that are independent of the attributes for the parameters and result. For example, Monotonic (Increasing) could be an attribute of a function; this might be of use in proving some property or performing a code optimization later.

5.5.4 Operator Declaration and Definition

Operator declarations specify an interface for an operator symbol, consisting of the operand and result types, and optionally attributes of the operator itself. An operator definition is (optionally) included in the local body of an operator declaration, and is associated with the interface. There may be more than one implementation associated with the same interface, but syntactically each implementation is associated with a distinct copy of the interface.

5.6 Expression Types

Expressions in HUSC consist of factors (literals, variables or function applications) and unary or binary operators applied to those factors. The type of an imported variable is assumed to be its declared type, although additional constraints may be

added to its type to support attribute assertions.

Literals have an automatic type judgement that includes a singleton range attribute, namely the value of the literal. The type is fixed by the parser when it is interpreted. More literal interpretations for multidimensional arrays should be added, or a constructor to build these out of linear arrays, remembering the arrays must be rectangular.

Variable types are derived from the definition of the variable, inferred from the use of the variable, and may explicitly declared. Where the type is declared, the derived (or inferred) type of the expression must be a subtype of the declared type.

5.6.1 Function Evaluation

The Coconut system accepts multiple possible implementations for each function evaluation, where each implementation must provide a semantically equivalent result. This is represented by a "joined" expression, where the value of the join is the value of any one of the member expressions, and each member expression is an implementation of an interface that is a supertype of the required interface. In other words, Coconut selects from all of the available interfaces that satisfy the type constraints demanded by the context of the function usage.

A function interface satisfies the constraints of a term if and only if:

- the arguments are subtypes of the parameters
- the result type, augmented by the propagation of attributes not specified in the interface, but included in either the argument types or as constraints on the term evaluation, satisfies all of the constraints on the term.

In other words, a function interface that does not specifically include an attribute class may still be used in a term where that attribute class is present, as long as the rules for attribute propagation prove that the implementation of the interface supports the propagation of the attribute class to the result.

No assumption is made that a single function interface satisfying the entire domain exists, nor that there will be any interfaces that satisfy the specified interface, if it is overly constrained. If no interface satisfies the constraints of the specification, a type error is issued.

5.6.2 Operator Evaluation

An operator consists of a syntactic symbol and a set of interface / implementation pairs. A symbol must have a constant precedence, fixity and arity to be used in an expression; infix, postfix, prefix are supported.

An operator interface consists of a type for each operand and the result. There may be multiple implementations, or no implementation, for a given interface within the context of a module; each implementation must be semantically equivalent. The value of an operator is the join of all of the implementations with satisfactory interfaces evaluated at it's operands, where the value of the join is the value of any one of the member expressions.

An interface is satisfactory if it satisfies all constraints on its result type and the operands are subtypes of the interface operand types. An operator interface satisfies the constraints on a term if and only if:

- the arguments are subtypes of the parameters
- the result type is a subtype of the meet of all of the constraints on the term
- the result type augmented by the propagation of attributes not specified in the interface, but included in either the argument types or as constraints on the term evaluation, satisfies all of the constraints on the term.

In other words, an operator interface that does not specifically include an attribute class may still be used in a term where that attribute class is present, as long as the rules for attribute propagation prove that the implementation of the interface supports the propagation of the attribute class to the result.

5.6.3 Domain Testing on Dimensioned Types

Domain and bounds checking of dimensioned types is part of the type checking / inferencing within HUSC. The index expressions are explicitly constrained to be integers within the range of the appropriate dimension of the domain. This works whether or not the domain is explicitly defined; where it is not explicitly defined, variables representing the dimension bounds are generated.

The formal rules for the bounds checking are built in to the definition statements, but testing that the partial domain do not overlap and do cover the full domain was not fully implemented. It is possible in some circumstances to verify that the set of partial definitions form a complete but unambiguous definition as a partition of the domain. Some checking of domain consistency has been implemented in the current system, but only where domains are adjacent ranges of integers, with the bound between ranges being numeric or using at most one common variable. For example in the following the partial definitions for A would be shown to be total, but for B this could not currently be verified.

```
A[0] = 0;  for j = 1..n-1 A[j] = j; A[n] = 0

B[0] = 0; for j = m..n B[j] = j; B[endval] = 0;
     m = 1; endval = n;
```

In order to deal with this issue, a greater level of symbolic computation would be required within HUSC. It should be possible to use (or adapt) existing constraint solvers to test the partitioning of a domain by subdomains, even where the bounds are defined symbolically.

When the domain cannot be verified as partitioned, there is an outstanding unresolved constraint indicating this failing. It is quite reasonable for a HUSC module to have such an unresolved constraint, especially where the domain is based on input definitions, but the compilation cannot be considered complete and well-typed until the context supplies sufficient information to conclude the definition is total.

5.7 Proof Obligations and Run Time Tests

This section provides an informal definition of how attributes may be asserted or tested for in the module context, attribute propagation rules, and how external proof obligations are tested or eliminated. The formal definition is provided in A.3.

The module context contains the definitions and type judgements for a module. The type judgements include subtype, shape and attribute judgements, where attribute judgements represent constraints that are demonstrably satisfied by an expression or definition. Where this proof cannot be derived from the context, based on

the typing rules or attribute propagation rules, the proof must be supplied in some other way.

5.7.1 Run Time Testing of Constraints

The attribute class definition may include a run time test function, a Boolean valued function that is True if the tested expression satisfies the constraint of the attribute. Run time tests are an obligation on the Coconut system to incorporate run time checks on the value of the expression to ensure that the constraint imposed by the attribute is always met. Should the state fail to satisfy the constraint, an exception is raised and the computation results in $\perp$. If this test function is not supplied, satisfaction of the attribute may not be tested at run time. The Coconut code generator is free to implement run time tests in any way, but the obligation for correctness is on the system.

When a module imports definitions from another module, it inherits all of the run time test obligations associated with the imported identifiers. Some of these may be resolved by the context of the importing module, in which case Coconut will not incorporate the run time tests in the generated code.

5.7.2 External Proof Obligations

Proof obligations represent constraints on expressions that were not derived or tested within the HUSC program. In order to consider the use of the program safe and accurate, the user of the program must be satisfied that all of the proof obligations are met. Obligations can be met by specification, testing, (non-Coconut) proofs or any other strategy that satisfies the user. No checking will be performed to ensure this is the case. Each module in Coconut will have a set of proof obligations, a detailed specification for its safe use.

When a module imports definitions from another module, it inherits all of the proof obligations associated with the imported identifiers. Some of these may be resolved by the context of the importing module, others may become part of the proof obligations of the new module. When a program is generated from the module the remaining proof obligations are made available as an associated file,

Attribute Consumption and Propagation

The interface to a function specifies the attributes that must be part of the parameter and result types. These result attributes are considered introduced by the function interface, while these parameter attributes, including the attributes of any dependent type expressions, are considered to be consumed by the function interface. The type system then attempts to prove the result attribute satisfaction based on the function implementation and the types of the terms it is applied to. If this is unsuccessful, and no run time testing was requested, a proof obligation is generated for the function evaluation for those terms. Note that since the actual argument may be a subtype of the parameter type, the specified attribute values may represent tighter constraints than necessary for the interface, and therefore may have fewer proof obligations than the interface in general.

When the argument type of the term contains attributes of a class not specified by the interface, or when the type of the term is constrained with an attribute not part of the result interface. these attributes may be propagated. Attribute propagation is governed by rules included in the attribute definition, and is based on pattern matching over typed expressions.

Chapter 6

Type System Implementation

There are four main modules to the type system. The main module is Constraint.pl is a CHR module, which is written and compiled in Prolog, but uses an extensive extension to Prolog's semantics. The last three (subtype,attribute and context.pl) are ordinary Prolog files, and are included by constraint.pl.

This chapter includes an overview of

- CHR semantics
- the internal representation of types
- the constraint handling rules implementation of the type semantics
- subtype implementation
- attributes
- managing the internal context, arithmetic and logical operations

This is followed by the detailed literate source code with documentation for each module.

6.1 Constraint Handling Rules

CHR is a term rewriting system in which the terms are a constraint name and a list of arguments, either variables or constants. The rules consist of a head, which is a

pattern of one or more constraints, a Boolean valued guard, and a body of constraints that will be added when the rule is fired. The system searches the current constraint context for matches for each rule, unifying the variables in the rule with those of the stored constraints, then executes the rule if the guard is satisfied. This continues until no more rules can be executed, or the system generates a failure condition.

There are three kinds of rules:

propagation just creates new constraints, leaves the existing constraints in the head

simplification deletes all of the constraints in the head

simpagation a combination where a specified subset of the constraints of the head are preserved, and the rest are removed; this is redundant but improves legibility (and offers a small optimization)

Once a constraint has been tested for its applicability in a specific rule, a token is generated to avoid having it tested again until there is a change. This allows the system to terminate only after testing all rules against all remaining constraints. However, if a constraint contains a free variable argument, and that variable becomes ground through its participation in another constraint, then the tokens are erased and the constraint again tested.

The rules are tested in a term first order, meaning the first constraint from the goal is tested against all of the rules before the next constraint is considered. If new constraints are generated, they are immediately tested in the order introduced against the rules. The rules are tested in top to bottom sequence. While the CHR system on its own is confluent, and it is hoped the HUSC type system is too, the performance of the system can be dramatically improved by having the rules fire in an appropriate order.

The full definition of the CHR system is available in [6].

6.1.1 Soundness and Termination Properties of CHR

The CHR subsystem is confluent, sound and complete, when the rule system implemented in CHR is confluent, sound and complete. While necessary, there are no proofs that the HUSC system exhibits any of these properties.

In [13], Sulzmann describes an extensible Hindley/Milner type system called HM(X), where X represents additional constraints. He implements it in CHR, but more importantly proves soundness and completeness for the HM(X) type system, given the restriction that each rule in X has a single constraint in the head.

The HUSC type system does not satisfy this rule, as many of the rule heads contain more than one constraint. However, there are two specific types of multiheaded rules that are used that might be more manageable for soundness proofs:

1. Rules where the head consists of only one dynamic constraint, and a set of static constraints, meaning constraints that are never altered by the constraint system. Since these are semantically identical to rules where the existence of the constraint is transformed to a guard looking for a fact, it is likely they will not pose a problem.

2. Strengthening rules take two like constraints and produce one that meets both conditions, replacing the original two. This is what the join and meet rules do, and this special type of rule will likely pose no problems to soundness.

This is just conjecture at this point, but it does lend confidence in the search for soundness and completeness proofs.

Confluence is of particular concern. A type judgement for a variable cannot be complete until all of its constraints are identified. CHR's operation semantic when faced with a multi-constraint goal is to fire all rules against the existing constraint store, until these are exhausted, then pull in the next constraint from the goal. Rules are tested in the order they are presented in the file, which does help somewhat.

While ideally HUSC would present the variable definitions in the top down order of the graph, it is impossible to guarantee that all constraints are present until the goal is completely loaded. To handle this, when a new constraint is encountered for a variable that has a type judgement, and the type judgement does not satisfy the constraint, it is erased, the new constraint is absorbed into the total constraints on the definition, and the type judgement process begins again (because the updated new constraint causes it to begin again). The very limited testing of this scenario has shown it to be successful, but further testing would be advised, perhaps as part of proving the confluence properties of the system.

6.1.2 Representing HUSC Type Semantics as CHR Rules

Note that definitions and expressions are implemented as CHR constraints instead of Prolog predicates, but could have been implemented either way. Consider a CHR simpagation rule acting on a static fact lit(a,Int). Then the first rule, with lit a "static" constraint,

```
lit(a,Int) \ chr1(b,c) <=> chr2(b)
chr1(b,c) ==> lit(a,Int) | chr2(b)
```

can be replaced with the second where a is now a Prolog fact. Both are loaded with the definitions module, both delete chr1(b,c) and add chr2(b), the only difference is that the rule selection process within CHR. It was not clear which approach would be better, but a performance hint from the CHR site suggested avoiding guards where possible, so the former approach was selected.

6.1.3 Representing HUSC Types in Prolog

Types are represented as the predicate t/3 in the type system, with t/2 being the name and shape of a type without its attributes. Types consist of a name, list of dependent variables wrapped in the internal term predicates, and for t/3 the list of attributes as internal attribute representations. Pure generic HUSC shapes are encoded in the same fashion, with the name of the type being the name of the shape (e.g. Function).

Dependent variables are either type terms or data terms, represented by tt and dt predicates respectively. A type term is either a type (t) or a type variable with a supertype $(tv, t) \implies tv \preceq t$. Data terms are in the form dt(pt, val), where pt is one of the internal types: int, float or boolean, or an interval (val1,val2) or list [val1, val2, ...] of the above, and val is either a literal or a single variable instance.

Attributes are represented by attr(class, property), where the class name is an atom and the property is encoded in a data term. Note that this restricts the properties of attribute classes to integer, float, boolean and intervals or lists of these.

Conventions define the makeup of the dependent variable lists for function and dimensioned shape descriptions. These are important both as examples of the internal

type description, and necessary to understand the underlying resolution mechanism as it applies to shapes.

t(Function, [tt(return type), dt(num parms), tt(parm1type), tt(parm2type)...])

t(Array, [tt(element type), dt(rank), dt(interval(int), rng1), dt(interval(int), (l,u))...])

It is clear here that an array is a subtype of a function, but with a domain guaranteed to consist of the cross product of integer intervals. This subtype relationship may even be included in the array prelude file, if desired, although traditionally scientific programmers view arrays as storage structures not functions. It might be interesting to model linear operators as arrays and explore the subtype relationship between matrices and functions.

The internal representations, along with some simple arithmetic and logic functions, form the basis of a rudimentary symbolic computation system that allows dependent type variables and attributes to be evaluated. This system would not sufficient for production use, nor has it been well tested, but it is sufficient to verify the approach taken. Ideally, a full symbolic expression evaluator would be incorporated to manage the base arithmetic, and a HUSC interpreter would be used to generate the base level code for dependent variables and attributes with properties that are more complex HUSC types. A design and implementation for this would be a significant challenge, and restrictions may have to be placed on the types of dependent variables and attribute property types.

It is not necessary for named types to be stored in the same format as their underlying shapes, and in fact it is not even necessary for any of the shapes to exist, although HUSC without a function shape would be a mere shell.

6.1.4 Pseudo-literate Prolog

SWI-Prolog does not support literate source code. A quick and dirty solution to this problem is to use the following approach:

```
/* \begin{verbatim}
... latex ...
*/ %\end{verbatim}
```

(without the % in front of end), which is quite simple and effective, but does leave "/*" and "*/" scattered throughout the documentation. There are a number of generic packages for supporting general literate code writing, but this will suffice.

6.2 Implementation Overview

This section briefly describes each of the four modules that make up the type system. This includes key implementation decisions, observations and issues. It also serves as a guide to allow selective browsing of the detailed information in the modules themselves.

6.2.1 Constraints

This is the main module for the type system. There are a number of subsections in this module.

Main Clauses the goals to load the generated type and definitions files then run the type system

Constraint Generation CHRs to generate shape and attribute constraints from the module definitions

Constraint Resolution CHRs to resolve shapes and attributes from function interface definitions and attribute propagation rules.

Attribute Rules CHRs that define joins, meets and propagation rules for attribute classes. These are supposed to have been generated from the HUSC attribute class definitions, but that has not been done yet. For now, the rules for Range and Symmetric are hard coded.

Constraint Arithmetic CHR solvers for Boolean and some simple Integer arithmetic. Used to support the attribute propagation, join and meet rules. These are based on CHR solvers from K.U.Lueven CHR web site.

These should really be multiple modules, but it appears that CHRs must all be in the same module, or at least that a constraint invoked in an included module does not become available in the constraint system in the including module. More work in this area, including contacting the authors of the CHR tools, should be done as the current module is too large and unnecessarily confusing.

6.2.2 Subtype, Join and Meet Predicates

This module handles the join, meet and subtype clauses for named types and shapes with dependent variables. Shape rules are hard coded in HUSC, and generally force shapes to be identical to be compatible. Named types explicitly define their supertype, which is stored in the explicitly subtype hierarchy.

The join implementation has one small twist that might be confusing. The join of two types with dependent variables is the join of the resulting type, namely the first dependent variable in the representation, and the meet of all of the remaining terms, with the meet of every data term being equality or failure (bottom). This is effectively saying that the join of functions with typed parameters is contravariant in its relationship with the joined function parameter types, and covariant in the result type. This might seem at odds with the definition of a function interface, which is covariant in both result and parameters, but the function interface is a parameterized data definition, not a type definition, and represents a specialization of the function class and must be explicitly supported by the type context it is used in.

The process of evaluating these relations is largely managing an internal context for dependent variable expressions, resolving equality, bounds (less / greater than) and sub / super type relationships. This is handled in the context module, described here 6.2.4

6.2.3 Attribute Predicates

This module contains the generated predicates for the attribute class specific sub-attribute relation. It is currently hard-coded with the predefined attributes, but is intended to be generated from HUSC modules.

It also contains mechanics to evaluate a list of subattribute relations in a common context, which is required to support the subtype relation implementation.

6.2.4 Context Predicates

This module contains some very mechanical Prolog predicates to manage contexts for dependent variables. These support equality, bounds (lt,gt) and sub/super type relations between expressions (atoms).

A function / operator interface satisfies a type context if and only iff:

1. the interface result type is a subtype of the result expression type

2. each argument expression type is a subtype of the matching interface parameter type

There is a significant implementation detail in this assessment; any shape or named type judgement (et) that includes dependent type arguments must be true in a shared data context with the other dependent arguments. For example, in this example

```
module example
  export(C:: Array[-n..5] of Real)
  import(n::Integer)
{
Let A be Array[-10..n] of Real;
Let B be Array[-10..5] of Real;
A = ... ;
B = ...;
C = A + B;
}
```

The addition operator for arrays demands that the dimensions of the operands and result be identical. Testing the arguments A and B against the operand types will require the result to be of dimension -10..5, accomplished by letting n=5. Testing the result type will require the answer to have dimensions -n..5, accomplished by allowing the result type to be -10..5, with n as 10. Since this is not possible, the type of C should be reduced to type bottom to flag the error.

This problem is resolved by testing the result and parameter types in the Prolog predicate matchlist, from the context.pl file. This maintains a local dependent variable context, for integer, boolean, float and type variables. Equality, less / greater than and subtype relations are managed in the context, and if the context cannot be unified, the predicate reports a failure. This failure stops the function / operator interface candidate from being incorporated into the join, leaving the join candidate.

Note that the context management can also be used for the module data context when that is implemented. The global context contains literal and imported data items of type Integer, Real, Boolean, and type variables with their associated supertype. As computations requiring dependent variables be updated, this global context is passed into the local context, and then any modifications to the global values are pulled from the output local context.

It is unclear whether the Prolog system or the local CHR subsystem should be used to judge module level constraints on data items. The Prolog system is more complicated, and will likely have more difficulties with very complex systems, but provides immediate response to failed constraint unification that can prevent an invalid decision and clearly identify the context of the error. The CHR system will maintain the data and type variable constraints for very complex systems easily, but once a context error is identified, the system has already invoked the rule that generated the error. The system must then either just fail, or actively reverse the previous decision. Additional work will be required to resolve this decision, and this has been left to the future work category [1].

6.2.5 Summary

The fundamental structure of the type system worked well over the very limited range of examples that were tried. The Constraint Handling Rule language was a near perfect fit for the lattice structure of the type system. The main problem with CHRs was the inability to use multiple modules, and there might be a work around to that problem (or future releases might support this as a new feature). Otherwise the implementation was straightforward.

[1] Implementing these approaches would not be very time consuming, but validating their correctness will be.

Although there are a substantial number of Prolog predicates in use, translating the system to a functional programming language like Haskell would probably make the development of a production strength system easier. Almost all of the Prolog predicates are used as functions, with the ability to backtrack used only in resolving a types base type using its dependent arguments. There are a significant number of cuts to prevent the code from wasting time backtracking needlessly.

The biggest problem was in managing the variable context during dependent argument inferencing. This was managed for the limited internal types (int, float, boolean, type variables) in the local contexts fairly well, ensuring that a list of subtype judgements were internally consistent. Then ensuring that any values or constraints assigned to declared variables that appear in a dependent variable expression are used consistently throughout the module evaluation would require maintaining a module context for all data terms of integer, real or boolean type.

It might be worth re-examining implementations of CHRs in Haskell. CHR should be straight forward to implement in Haskell, it fits the state monad model very well and the semantics are quite clear. There is an implementation currently available that might be sufficient to manage the type system, this could be explored further. If it were in Haskell, the internal context could be managed by a full HUSC interpreter, which could then support all HUSC types that had sufficient definitions to support them as part of the type system (perhaps with a bit of bootstrapping). That approach would simplify the code and make proving properties about (and maintaining) the system far easier, while also eliminating the burden of translating the data structures too and from Prolog.

/*

6.3 CHR Type Inferencing

This is the main module in the implementation of the HUSC type system as a Constraint Handling Rules (CHR) program. It uses the CHR library implemented in SWI-Prolog (and compatible with SICtus Prolog) by K.U.Leuven, Belgium, `http://www.cs.kuleuven.be/~toms/CHR/index.html`.

All of the constraints are in the main module, because it appears that constraints

can't be used across module boundaries. This seriously reduces the clarity of the algorithms, and may be a misunderstanding, so should be reexamined in the future.

It loads Prolog predicates from the following modules:

subtype predicate implementing predicate relation.

attribute predicates implementing subattribute relation for individual attribute classes, and some generic subattribute relations (e.g. $A(\rho) \preceq^A A(\rho)$).

context a set of predicates that manage a list based local variable context used when determining if a function arguments are a subtype of the function interface, resulting in multiple subtype evaluations over a common local expression context.

```
 */
:- use_module(library(chr)).
:- use_module(subtype).
:- use_module(context).
:- use_module(attribute).
/*
```

6.3.1 Unavailable Prolog options

The following options were extracted from the constraint solvers incorporated into the type solver. They are not implemented in the SWI implementation, but have been included in the documentation in the event a Sictus implementation is considered.

- :- handler husctype % from Sictus, necessary to run WebCHRs.
- :- chr_option(already_in_heads,on). % useful optimization for replacing constraints
- :- option(check_guard_bindings, on). % for = /2 with deep guards.

6.3.2 Operator Definitions

These operators are used in some of the CHR handlers for arithmetic and max / min problems.

```
 */
:- op(700, xfx, lss). %% less than
:- op(700, xfx, grt). %% greater than
:- op(700, xfx, neq). %% not equal to
:- op(700, xfx, geq). %% greater or equal to
:- op(700, xfx, leq). %% less or equal to
:- op(700, xfx, ~=). %% not identical
/*
```

6.4 Main Constraints

This section includes the three main processing goals, and lists all of the predicates that are to be treated as constraints by the CHR subsystem. The CHR system is implemented using attributes associated with the predicates to maintain an extensive information store with each predicate and predicate argument. This declaration is mandatory, and any predicate name used as a CHR will not also be used as a standard predicate.

The main goal is *g*o, which loads the types (as clauses) from *modname*_types.pl and the clause *r*un from the module definitions file *modname*_defs.pl. These are the files generated by the HUSC parser, with *modname* being the name of the main program module. It then invokes the run goal, which consumes the constraints representing the module definitions and declarations, and evaluates type judgements for variables and join candidates for operator and function evaluations. It also tests type definitions to ensure each defined type is a subtype of its base type (even if the subtype relationship is explicitly cancelled), removing types that do not satisfy this from the store (and recording the error).

The *c*leanup predicate extracts the constraints remaining in the constraint pool after all rules have been tried. Significant information, namely

vt type judgements for variables

joinex operator / function interfaces selected to be part of an implicit join

xpfbol external proof obligations as an expression and attribute to be proven

error any errors logged in the process

is written to the *modname*_results.txt. A number of constraints will be left behind as a result of normal processing, and these are written to a log file *modname*_log.txt to support debugging HUSC programs. These are:

joincan operator / function interfaces rejected from a particular implicit joined expression.

attrule attribute rules that were pulled in because they matched an operator expression by symbol (or function name) but were rejected based on the types of the operands / parameters.

```
 */
:- chr_constraint
            et/2,ec/2, joinex/4,
            unop/3,lit/2,var/2, binop/4,fnapp/3,
            arraydef/3,fnimp/3,qualuniv/2,
            vdecl/2,vt/2,vdef/2,
            genecs/2,genacs/2,genobls/2,genattrcons/2,
            genattrules/2,attrule/3,
            ac/2,allacs/2,xpfobl/2,rttest/2,attrcon/2,
            attrjoin/4,attrmeet/4,
            genjoincan/3,joincan/3,
            cocon_and/3,cocon_or/3,cocon_xor/3,
            cocon_not/2,cocon_imp/2,cocon_eq/2,
            add/3,neg/2,
            (~=)/2, (leq)/2, (lss)/2, (neq)/2, min/3, max/3,labeling/0,
            error/2.
```

```
go(F) :- atom_concat(F,'_types.pl',TF),consult(TF),
         atom_concat(F,'_defs.pl',DF),readGoalFromFile(DF,G),
         asserta(G),run(DF).

readGoalFromFile(F,G) :-
    open(F,read,S,[eof_action(eof_code)]),
    read_term(S,G,[singletons(warning)]),
    close(S).

/* cleanup will eventually write the output files */
cleanup(F) :- write(F).

/*
```

6.5 CHR Constraint Generation

The first kinds of constraints are representations of the underlying definitions.

6.5.1 HUSC Constraint Predicates

HUSC generates constraints that restrict possible type judgements. Constraints for shape and each individual attribute class are issued and resolved separately. All of the individual attribute constraints applied to an expression are collected up in a list in order to process a variable length attribute list, this list is initialized to empty here for each expression. There may very well be a better strategy for this.

Note that there is an overloading to the word "constraint", which sometimes means a predicate managed by CHR subsystem, and sometimes means a true HUSC constraint

t(n,deps,attrs) is the internal representation of a HUSC type, with the type name n, a list of dependent variables, and a list of attributes. If there are dependent variables, the first MUST be the returned type as an expression of the dependent

variables over the type. If there are no dependent variables, the name of the type is the total type description and therefore the returned type. The remaining dependent variables are either data or type terms, their order and make up is shape dependent.

allacs(e,[ac1,..,acn)] expression e has ac1,..,acn attribute constraints

ec(e,t) expression e is constrained to be of shape t (no attributes); t is used in ec constraints throughout the code instead of s for anachronistic reasons

genattrcons(E,t) creates an attribute constraint for each attribute in type t

6.5.2 HUSC Factor predicates

Factors include literals and variables, expressions in parentheses have alredy been absorbed into the syntax graph. Note that HUSC variables are represented as Prolog atoms, not Prolog variables. Only dependent variables in type definitions are represented as Prolog variables, which are used to instantiate specific instances of the base / supertype of a named type with dependent parameters.

These are static constraints that are never modified or erased during processing of the CHR rules.

lit(e,t) expression e is a literal of HUSC type t (with attributes); a literal is constrained to be an very restricted type including attributes, e.g. the range of a literal integer of value n is (n..n)

var(e,var1) expression e is an instance of var1 shape and attribute constraints on an expression that is a variable instance are applied to the variable, and therefore to it's defining expression

vdecl(var1, t) var1 is declared to have HUSC type t, including attributes

vdef(var1,e) variable var1 is defined by expression e the constraints on the variable declaration are applied to the expression defining the variable

```
*/
lit(E,t(S,SAs,SAttrs)) ==>
   allacs(E,[]),et(E,t(S,SAs)),genattrs(E,t(S,SAs,SAttrs)).
vdef(V,E),vdecl(V,t(T,TAs,TAttrs)) ==>
   ec(E,t(T,TAs)),genattrcons(E,t(T,TAs,TAttrs)).
var(E,_) ==> allacs(E,[]).
var(E,V),ec(E,t(T,TAs)),vdef(V,E2) ==> ec(E2,t(T,TAs)).
var(E,V),vdef(V,E2),attrcon(E,A) ==> attrcon(E2,A).

/*
```

6.5.3 Operator and Function Applications

Operators and functions are the join of all possible interfaces that return a type satisfying the constraints on the expression and is satisfied by the operand / argument types. They are treated the same way, but have different CHR representations, and the number of operands is fixed while the arguments are in a list. This could be entirely replaced with the variable arguments list, but it is convenient to experiment on the operators first, without the extra level of indirection caused by processing the lists.

Again, these are static constraints that are never eliminated or modified during the rules processing.

unop(e,op,e2) expression e = op e2

binop(e,op,e1,e2) expression e = e1 op e2

unop(e,op,e2) expression e = f(e1,ℓ,en)

Operators and function applications are resolved by first getting the class definition of the op/func and applying those constraints to the result / operand types. Operator definitions are the result of HUSC operator definition statements; function class definitions from declarations of identifiers as function types. These are loaded as Prolog facts from the type definitions file.

The class definitions describe the most general usage of the operator symbol or function name. For operators, this is often completely open, or in HUSC terminology, mapping type top ($\top_T$) to top. For functions the constraints are usually more useful, for example sinusoids map a real to a real in the range (-1,1). These default constraints are applied to reduce the number of choices for argument expression type judgements.

The shape and attribute constraints from the operator and function class definitions are applied to the result and operand expressions. For functions, a new CHR constraint is created to apply the constraints from the generic parameters.

op1def(Op, rt, pt) defines the generic operator symbol $Op :: pt \to rt$

op2def(Op, rt, pt1,pt2) defines the generic operator symbol $Op :: pt1, pt2 \to rt$

fndef(F,rt,[pt1,..,ptn)] defines function class $F :: pt1 \to \ell \to ptn \to rt$

findall(Attr, reg_attrule(OpF, AttRule), AttRs) a Prolog predicate to retreive all of the attribute propagation rules that apply to a particular operator or function symbol.

genattrules(Attrs, [result expression, operand expressions)] propagates a list of attribute constraints to a list of expressions.

genjoincan(E,F,FIs) propagates a list of join candidates for function / operator F, function / operator interface FI.

Any attribute rules that match the expression are executed to infer new constraints on arguments / operands, or derive support for attributes (i.e. satisfies the constraint) of a result. This is the only way attribute constraints can be inferred for operands / function arguments; otherwise they must be explicitly declared in the function interface or declared / asserted as part of a variable definition.

Finally the list of available interfaces for the operator / function are searched looking for candidates for the join. This grabs all possible interfaces, their suitability given the context is determined later.

funcifs(F,FIs) collect all function / operator interfaces for the symbol or function name

genjoincan(E,F,FIs) wrap a CHR constraint around the interfaces as prospective join candidates for an expression.

Note that it is critical to evaluate the attribute rules first, meaning that genattrules must be invoked before genjoincan, so join candidates can be selected based on the new attributes derived / inferred by the rules.

```
 */
 unop(E,F,E1) ==>
    allacs(E,[]),op1def(F,R,P1),
    minsupt([],R,RC),minsupt([],P1,P1C),
    findall(AttR,reg_attrule(F,AttR),AttRs), funcifs(F,FIs)  |
      ec(E,RC),ec(E1,P1C), genattrules(AttRs,[E,E1]), genjoincan(E,F,FIs).
 binop(E,F,E1,E2) ==>
    allacs(E,[]),op2def(F,R,P1,P2),
    minsupt([],R,RC),minsupt([],P1,P1C),minsupt([],P2,P2C),
    findall(AttR,reg_attrule(F,AttR),AttRs),funcifs(F,FIs) |
      ec(E,RC), ec(E1,P1C), ec(E2,P2C),
      genattrules(AttRs,[E,E1,E2]), genjoincan(E,F,FIs).
 fnapp(E,F,Args) ==>
    allacs(E,[]),fndef(F,R,Parms),
    minsupt([],R,RC),minsupt([],Parms,PCs),
    findall(AttR,reg_attrule(F,AttR),AttRs), funcifs(F,FIs) |
    ec(E,RC), genecs(Args,PCs), genattrules(AttRs,[RC|PCs]),genjoincan(E,F,FIs).

 genattrules([],_) <=> true.
 genattrules([(AC,R)|AttRs],Es) <=>  attrule(AC,R,Es),genattrules(AttRs,Es).

 genjoincan(_,_,[]) <=> true.
 genjoincan(E,F,[FI | FIs]) <=> joincan(E,F,FI),genjoincan(E,F,FIs).

/*
```

Operator and Function Definitions

The op1def / op2def predicates find operator class definitions based on the symbol, and return their parameter types. If there is no specified class definition, the operator is assumed to be Top-¿Top or (Top,Top)-¿Top. i.e. it provides no constraint on it's operands or results.

The fndef predicate finds the matching function definition as a funclass predicate. The funclass predicates are function class type declarations; these are part of the module_types.pl along with the other type definitions generated by the HUSC parser. The confusion between function class declarations, function type definitions and function interfaces should be examined; there may be a cleaner way of handling these concepts.

Function and operator interface definitions are written by CocoTypes module into the type definitions file. All definitions for the operator / function name are returned in a list. The parameters of the funcintf predicate, representing a function interface, are:
funcintf(funcname, funcif name, return type, num parms (dt), list of parm types[tt(),...]

```
 */
% this is the default operator Top->Top, or Top->Top->Top.
funclass(t(cocoop1,[tt(t('Top',[],[])),dt(int,1),
                             tt(t('Top',[],[]))],[]),_,_,_).
funclass(t(cocoop2,[tt(t('Top',[],[])),dt(int,2),
              tt(t('Top',[],[])),tt(t('Top',[],[]))],[]),_,_,_).

op1def(F,R,P1) :- funclass(t(F,[tt(R),dt(int,1),tt(P1)],[]),_,_,_).
op1def(_,R,P1) :- funclass(t(cocoop1,[tt(R),dt(int,1),tt(P1)],[]),_,_,_).
op2def(F,R,P1,P2) :- funclass(t(F,[tt(R),dt(int,2),
                                                     tt(P1),tt(P2)],[]),_,_,_).
op2def(_,R,P1,P2) :- funclass(t(cocoop2,[tt(R),dt(int,2),
                         tt(P1),tt(P2)],[]),_,_,_).

fndef(F,R,Ps) :- length(Ps,N), maplist(wrapparm,Ps,TPs),
```

```
                    funclass(t(F,[tt(R),dt(integer,N), TPs]),_,_,_).
wrapparm(P,TP) :- TP = tt(P).


 funcifs(F,FIs) :- findall(FI,funcintf(F,FI,_,_,_),FIs).


/*
```

6.5.4 Attributes Constraints

Type declarations from HUSC have both shape and attribute constraints embedded in the same definition. This is decomposed into a shape constraint (ec) and then attribute constraints (attrcon) that force an expression to satisfy an attribute constraint. The list processing constraint genattrcons generates a constraint for each attribute in the type. Note that forcing an expression to support a constraint is not a type judgement nor does it indicate that the expression does satisfy the constraint the attribute represents.

When there are two attribute constraints of the same attribute class for the same expression, replace the attribute constraint with the "meet" of the attributes using the meet CHR rule defined for that attribute class (in this file). Named type / shape constraints are also merged with a meet, but this meet is a Prolog predicate from the subtype.pl file, which handles dependent type arguments and the explicit named subtype hierarchy.

Meets are idempotent, and many duplicate constraints get generated, so strip out identical duplicates, regardless of attribute class.

A newly defined type has an optional set of attributes associated with it, but the type it is based on may also attributes. These attributes are inherited from the supertype, and are merged with the new attributes by meets when of the same class.

ec is a shape constraint on an expression.

genecs zips a list of expressions with a list of shape constraints.

```
 */
/*
 * Hierarchical attributes, disabled for now.
 * should probably make this part of the type definition validation anyways.
 * genattrcons(_,t(top,[],[])) <=> true.
 * genattrcons(E,t(S,SAs,[])) <=>
 *    type(t(S,SPs),t(T,TAs,TAttrs),_,_),matchlist(SAs,SPs) |
 *      genattrcons(E,t(T,TAs,TAttrs)).
*/

 genattrcons(_,t(_,_,[])) <=> true.
 genattrcons(E,t(T,TAs,[A|As]))
     <=> attrcon(E,A),genattrcons(E,t(T,TAs,As)).
 attrcon(E,A) \ attrcon(E,A) <=> true. %% absorb duplicate
 attrcon(E,attr(AC,V1)),attrcon(E,attr(AC,V2))
     <=> attrmeet(AC,V1,V2,VM) | attrcon(E,attr(AC,VM)).

 genecs([],[]) <=> true.
 genecs([A|As],[P|Ps]) <=> ec(A,P),genecs([As|Ps]).
 ec(E,T) \ ec(E,T) <=> true. %% absorb duplicate
 ec(E,t(T,TAs,[])) <=> ec(E,t(T,TAs)).
 ec(E,t(T,TAs,TAttrs))
     <=> (not(TAttrs=[])) |
            ec(E,t(T,TAs)),genattrcons(E,t(T,TAs,TAttrs)).
 ec(E,T1),ec(E,T2) <=> meet(T1,T2,S) | ec(E,S).

/*
```

6.6 Constraint Resolution

This is the type judgement part of the system. At this point in the file, constraints have been generated and merged with meets. Now we find types, shapes and at-

tributes that we are sure of, and propagate them back up the expressions to make sure constraints are satisfied. Named type / shape / dependent arguments for expressions are encapsulated in an expression type judgement (et), with attribute support being separate (ac). For variables in a variable type judgement (vt), with the attributes bundled into the type description.

ac(E,attr) indicates expression E supports the attribute instance attr

et(E,t) indicates expression E is shape t or a supertype of t

vt(V,t) at the end of processing, this indicates variable V is of type t (with attributes), multiple vts may be generated and joined during processing.

genacs(E,t) generates attribute supports (ac) for each attribute the type t.

allacs(E,attrs) is a list of all of the attributes supported by expression E.

Due to the operational semantics of the CHR subsystem, the genacs must come before the et to allow attribute assertions to be satisfied and eliminated before testing function interface suitability.

```
 */
 lit(E,t(S,SAs,SAttrs)) ==> genacs(E,t(S,SAs,SAttrs)),et(E,t(S,SAs)).
 var(E,V),vt(V,t(T,TAs,TAttrs)) ==> genacs(E,t(T,TAs,TAttrs)),et(E,t(T,TAs)).
/*
```

6.6.1 External Proof Obligations on Input Variables

Variables that are defined as inputs to module can absorb attribute assertions placed on them by translating them to proof obligations for the use of the module. This is equivalent to refining the type definition of the input definitions to the module, which of course are also a type of proof obligation. Ideally these new obligations should be added to the interface for the module, but this is not done automatically as there may be a reason for leaving them out. The shape and attribute constraints that are not supported by the type declaration of the input variable are generated as external proof obligations. Then the attribute support is announced (ac) in either case.

vdef(v, cocoinput(v)) the definition of v must be supplied by a module that imports this one

xpfobl(v,attr(A)) attribute A is an external proof obligation for variable v for this module.

genobls generates a list of proof obligations from a list of attributes

The external proof obligations are generated lower in the module, in 6.6.2, after the satisfied attribute constraints are removed.

```
 */
 vdef(V,cocoinput(V)),vdecl(V,t(T,TAs,TAttrs)) ==>
   genacs(cocoinput(V),t(T,TAs,TAttrs)),
   et(cocoinput(V),t(T,TAs)),
   xpfobl(cocoinput(V),attr(shape,t(T,TAs))),
   genobls(cocoinput(V),t(T,TAs,TAttrs)).

/*
 * The following two lines do the hierarchical attributes,
 * I'm skipping this for now.
 * genacs(_,t(top,[],[])) <=> true.
 * genacs(E,t(S,SAs,[])) <=>
     type(t(S,SPs),t(T,TAs,TAttrs),_,_),matchlist(SAs,SPs) |
        genacs(E,t(T,TAs,TAttrs)).
*/
 genacs(_,t(_,_,[])) <=> true.
 genacs(E,t(T,TAs,[A|As])) <=> ac(E,A),genacs(E,t(T,TAs,As)).

/*
 * The following two lines do the hierarchical attributes,
 * I'm skipping this for now.
 * genobls(_,t(top,[],[])) <=> true.
 * genobls(E,t(S,SAs,[])) <=>
```

```
 *    type(t(S,SPs),t(T,TAs,TAttrs),_,_),matchlist(SAs,SPs) |
 *       genobls(E,t(T,TAs,TAttrs)).
*/
 genobls(_,t(_,_,[])) <=> true.
 genobls(E,t(T,TAs,[A|As])) <=> xpfobl(E,A),genobls(E,t(T,TAs,As)).

/*
```

6.6.2 Satisfying Attribute Constraints

Recall that the ac constraint indicates an expression satisfies an attribute, and the attrcons constraint indicates the expression must satisfy the attribute. This next group of rules eliminates satisfied constraints (attrcons) where possible.

An attribute (ac) removes an attribute constraint (attrcon) if it is at least as constraining, meaning it is a subattribute. The subattribute relation is defined in the attribute class definition, and encoded as a Prolog predicate in the subtype.pl module.

If there are multiple attributes of the same class on the same expression, then the expression is determined to satisfy the join of the attributes. This typically happens over join expressions, where the each of the joined interfaces returns a result that satisfies a slightly different constraint. The join (or least general generalization) is then determined by the attribute class definition, given the two attribute property values. For example, if the application of a real valued function results in two possible interfaces for the function, returning result in the range (1.0 .. 2.0) and (1.3 .. 2.1) respectively, then the range of the result is in range (1.0 .. 2.1), because either of the joined interfaces could be used.

The attribute join constraints for all attribute classes are currently stored in this module because all CHR rules must be in one module. This is not particular desirable, because these are intended to be generated from HUSC modules, and no other code in this module is generated. However, since all of the available attribute CHRs and predicates are currently being hard coded this is not a problem at the moment.

Note that identical attribute supports (ac) are removed first.

attrjoin(AC,x,y,z) get the join of the values (attr(AC,z)) for two attributes of the same class from the class generated rules.

subattr(AC,x,y,_,_) interprets the subattribute relation for attribute class AC, part of the class generated code (the extra parameters are used for context in recursive calls to subattr).

The generation of external proof obligations for new attribute constraints applied to input variables, as described in 6.6.1, is handled here after the satsified constraints are removed.

```
 */
 ac(E,A) \ ac(E,A) <=> true.
 attrcon(E,attr(AC,V1)) \  ac(E,attr(AC,V2)) <=>
    not(subattr(AC,V2,V1,_,[])), attrmeet(AC,V1,V2,VM)
     | ac(E,attr(AC,VM)).
 ac(E,attr(AC,V1)), ac(E,attr(AC,V2))
     <=> attrjoin(AC,V1,V2,VJ), ac(E,attr(AC,VJ)).
ac(E,attr(AC,V1)) \ attrcon(E,attr(AC,V2))
    <=> subattr(AC,V1,V2,_,[]) | true.
 ac(E,attr(AC,V1)) \ allacs(E,As)
     <=> not(member(attr(AC,_),As)) | allacs(E,[attr(AC,V1)|As]).
 ac(E,attr(AC,V1)) \ allacs(E,As) <=>
    member(attr(AC,V2),As), not(subattr(AC,V1,V2,_,[])),
    delete(As,attr(AC,_),OAs), attrjoin(AC,V1,V2,VJ)
     | allacs(E,[attr(AC,VJ)|OAs]).
% actually, this join should be redundant, should already have joined the ac's

 allacs(cocoinput(E),As), attrcon(cocoinput(E),attr(AC,Val)) <=>
    not(member(AC,As)) |
    ac(cocoinput(E),attr(AC,Val)),xpfobl(cocoinput(E),attr(AC,Val)).

/*
```

6.6.3 Resolving Test and Proof Obligations

An attribute (ac) will also remove the need for a run time test(rttest), and external proof obligation (xpfobl) in the same way as it removes attribute constraints.

In the same way, the inclusion of a runtime tests satisfies an attribute constraint or an external proof obligations.

rttest(E,A) expression E is tested at run time to assure it supports A

xpfobl(E,A) attribute A must be supported by E is an external proof obligation for the module.

Note: one pf obligation may spawn many assertions through inference rules.

Note the first two rules don't apply to input (imported) definitions, which were handled in 6.6.1.

```
*/
ac(E,attr(AC,V1)) \ xpfobl(E,attr(AC,V2)) <=>
   (not(E=cocoinput(_))),subattr(AC,V1,V2,_,[]) | true.
ac(E,attr(AC,V1)) \ rttest(E,attr(AC,V2)) <=>
   (not(E=cocoinput(_))),subattr(AC,V1,V2,_,[]) | true.
rttest(E,attr(AC,V1)) \ attrcon(E,attr(AC,V2)) <=>
   subattr(AC,V1,V2,_,[]) | true.
rttest(E,attr(AC,V1)) \ xpfobl(E,attr(AC,V2)) <=>
   subattr(AC,V1,V2,_,[]) | true.
rttest(E,attr(AC,V1)), rttest(E,attr(AC,V2)) <=>
   attrmeet(AC,V1,V2,VJ) | rttest(E,attr(AC,VJ)).
rttest(E,A) \ allacs(E,As) <=>
   not(member(A,As)) | allacs(E,[A|As]).
xpfobl(E,attr(AC,V1)),xpfobl(E,attr(AC,V2)) <=>
   attrmeet(AC,V1,V2,VJ) | xpfobl(E,attr(AC,VJ)).

/*
```

6.6.4 Selecting Function / Operator Interfaces for Joins

Operator / function interfaces are selected based on satisfying the type constraints applied to the result and operand expressions. This requires type judgements (et, allacs) for each operand / argument to be available. Each join candidate is tested, adding each one that satisfies all remaining attribute constraints to the join expression. Note that this occurs after the attribute rules have fired, and any attribute constraints that are satisfied independently of the selected implementation of the operator or function are removed. The attribute propagation rules are covered in 6.7 and the satisfaction of attribute constraints in 6.6.2.

The matchlist predicate, from context.pl, tests all of the parameter and result subtype relations in a common dependent type variable context. If this predicate is satisfied, all of the subtype relationships are true and the dependent type arguments are consistent. In addition, data terms will be expressed in their most specific form; that is if a data term is equal to n and 10, it will be presented as 10.

It is possible that constraints introduced later will cause and expression type to be further constrained than it is when these rules are fired because of the introduction of new constraints from later on in the HUSC module. None of the acceptable judgements will be undone by this, as the new type can only be more constrained than the previous one. In addition, updating the expression type judgement (et) will cause the rules to fire again on the remaining unsuccessful join candidates still in the constraint pool, and pick up any that are now satisfied by the new, tighter constraints.

Note that in this implementation, only functions of up to three parameters are supported. This is a mechanical issue - there needs to be a constraint that represents when all of the arguments have a type judgement. Otherwise the lists can be mechanically expanded by cut and paste, noting that operators and function applications are identical in resolution.

```
*/

unop(E,F,E1),et(E1,S1),allacs(E1,A1s),ec(E,T)
       \ joincan(E,F,FI)
 <=> funcintf(F,FIN,tt(IR),dt(int,1),[tt(IP1)]),
```

```
     setattr(S1,A1s,SA1s),
     matchlist([tt(SA1s),tt(IR)],[tt(IP1),tt(T)],CTX,[]),
     minsupt(CTX,IR,IRC)
        | joinex(E,FIN,IRC,CTX).
binop(E,F,E1,E2),et(E1,S1),et(E2,S2),allacs(E1,A1s),allacs(E2,A2s),ec(E,T)
       \ joincan(E,F,FI)
     <=> funcintf(F,FIN,tt(IR),dt(int,2),[tt(IP1),tt(IP2)]),
         setattr(S1,A1s,SA1s),
         setattr(S2,A2s,SA2s),
         matchlist([tt(SA1s),tt(SA2s),tt(IR)],[tt(IP1),tt(IP2),tt(T)],CTX,[]),
         minsupt(CTX,IR,IRC)
         | joinex(E,FIN,IRC,CTX).
fnapp(E,F,[E1]),et(E1,S1),allacs(E1,A1s),ec(E,T)
       \ joincan(E,F,FI)
 <=> funcintf(F,FIN,tt(IR),dt(int,1),[tt(IP1)]),
     setattr(S1,A1s,SA1s),
     matchlist([tt(SA1s),tt(IR)],[tt(IP1),tt(T)],CTX,[]),
     minsupt(CTX,IR,IRC)
        | joinex(E,FIN,IRC,CTX).
fnapp(E,F,[E1,E2]),ec(E,T)
         ,et(E1,S1),allacs(E1,A1s)
         ,et(E2,S2),allacs(E1,A2s)
       \ joincan(E,F,FI)
     <=> funcintf(F,FIN,tt(IR),dt(int,2),[tt(IP1),tt(IP2)]),
         setattr(S1,A1s,SA1s),
         setattr(S2,A2s,SA2s),
         matchlist([tt(SA1s),tt(SA2s),tt(IR)],
                             [tt(IP1),tt(IP2),tt(T)],CTX,[]),
         minsupt(CTX,IR,IRC)
         | joinex(E,FIN,IRC,CTX).
fnapp(E,F,[E1,E2,E3]),ec(E,T)
         ,et(E1,S1),allacs(E1,A1s)
```

```
          ,et(E2,S2),allacs(E2,A2s)
          ,et(E3,S3),allacs(E3,A3s)
        \ joincan(E,F,FI)
      <=> funcintf(F,FIN,tt(IR),dt(int,3),[tt(IP1),tt(IP2),tt(IP3)]),
          setattr(S1,A1s,SA1s),
          setattr(S2,A2s,SA2s),
          setattr(S3,A3s,SA3s),
          matchlist([tt(SA1s),tt(SA2s),tt(SA3s),tt(IR)],
                    [tt(IP1),tt(IP2),tt(IP3),tt(T)],CTX,[]),
          minsupt(CTX,IR,IRC)
          | joinex(E,FIN,IRC,CTX).
/*
```

6.6.5 Expression Type Judgements

At this point, type judgements for the expression are issued in the form of expression type judgements (et) for named types / shapes / dependent arguments, and attribute supports (ac). For variable instances, the type of the variable is are also the expression, including attributes.

For input variables, first any remaining attribute constraints are added to the external proof obligations, then the attribute supports (ac) are issued for all of the attributes in the input type declaration and those required by attribute constraints.

For function / operator applications, each successful join candidate (joinex) now submits its result type as a type judgement for the result expression, including any attributes. These are then joined to produce the final expression type judgement. The attribute class rules are fired before these rules are tested, so any attribute constraints that were satisfied by implementation independent rules no longer restrict the type of the result.

When multiple type / shape judgements (et) for an expression are generated, they are merged with the join predicate from the subtype.pl file. This join handles dependent type arguments and the explicit subtype hierarchy.

Note that when propagating the attributes guaranteed by the successful join candidate, the attribute supports (ac) must come before the expression type judgement

(et) to clear out any attribute constraints that would conflict.

```
 */
 ac(E2,attr(S,Val)),var(E,V),vdef(V,E2) ==> ac(E,attr(S,Val)).
 attrcon(cocoinput(E),attr(S,Val)) ==>
     ac(cocoinput(E),attr(S,Val)),xpfobl(cocoinput(E),attr(S,Val)).

 joinex(E,FI,R,CTX) \ joinex(E,FI,R,CTX) <=> true.
 ec(E,T),attrcon(E,A) \ joinex(E,_,IR,CTX) <=>
     setattr(T,[A],TAs),not(st(IR,TAs,_,CTX)) | true.
 joinex(E,_,t(T,TAs,TAttrs),_) ==> genacs(E,t(T,TAs,TAttrs)),et(E,t(T,TAs)).
 ec(E,T),attrcon(E,A) \ et(E,S) <=> setattr(T,[A],TAs),not(st(S,TAs)) | true.
 et(E,S1),et(E,S2) <=> join(S1,S2,T) | et(E,T).
/*
```

6.6.6 Variable Type Judgements

Variable judgements are the result of an expression type judgement on the expression defining a variable. If there are multiple variable type judgements, they are joined, after first stripping out duplicates.

The second rule flags variable type definitions that have a constraint that is not satisfied, which means a new constraint was generated from definitions later in the module definition goal. The variable type judgement is erased; the expression type judgement will also be erased (above), and the process will begin again. In theory this is highly unsatisfactory from a performance perspective; it is hoped that these problems can be minimized by an effective ordering of variable declarations and definitions in the module description file.

```
 */
 vdef(V,E),et(E,t(T,TAs)) ==> vt(V,t(T,TAs,[])).
 ec(E,T),attrcon(E,A),vdef(V,E) \ vt(V,S) <=>
    setattr(T,[A],TAs),not(st(S,TAs)) | true.
 vt(V,S) \ vt(V,S) <=> true.
```

```
vt(V,t(S1,S1A,_)),vt(V,t(S2,S2A,_)) <=>
   join(t(S1,S1A),t(S2,S2A),t(T,TA)) | vt(V,t(T,TA,[])).
vt(V,t(T,TAs,TAttrs)),vdef(V,E),ac(E,A) <=>
   (not(member(A,TAttrs))) | vt(V,t(T,TAs,[A|TAttrs])).
```

```
/*
```

6.7 Attribute Class Definitions Part I

Each attribute class must supply a definition of a join, meet and subtype relation. In addition, the class may specify propagation rules that define derivation and inference rules for the class. These rules are pattern matched over function names and operator symbols, and then type checked to make sure they are appropriate.

The rules are given as CHRs, propagating attribute constraints and supports over the arguments and results of operator / function applications. Any data terms are provided in the HUSC internal data and type terms (dt , tt resp.), which forms part of a small context management capability. This aspect is of the system is fairly weak, as actually building in full symbolic computation was beyond the scope of this project. More about this internal system is given in 6.2.4.

This is part I of the definitions because only the CHR definitions for the attribute class are in this file. The remainder are in the attribute.pl file.

For this implementation, two attributes have been hard coded to represent some of the different kinds of attributes that might be defined.

6.7.1 Symmetric

A dimensioned type is symmetric if it has rank two, it's two dimensions are identical, and it equals its transpose (the traditional symmetric matrix definition). This is a good example of a Boolean valued attribute, with the unknown state being represented by the absence of the attribute.

The join relation is equivalent to Boolean and, because all elements of a join expression must return symmetric arrays if the result is to be guaranteed symmetric.

The meet is an or, as if there are two contexts for the data item, and either one requires the shape to be symmetric, then the shape must be symmetric. The subtype relation requires that the subtype be symmetric if the supertype is symmetric, but only then.

Adding and negating symmetric arrays produces a symmetric array. Pointwise scalar multiplication of a symmetric array is a symmetric array.

The meet,join and subattribute relations and many of the rules presented here are clearly applicable in the case of many properties that might be of interest (tridiagonal, banded, positive definite). Generallzing these properties has been left for future work.

```
 */
/* general property of all join / meet relations */
 attrjoin(_,V,V,VJ) <=> VJ=V.
 attrmeet(_,V,V,VM) <=> VM=V.

attrjoin(symmetric,V1,V2,VJ)
    <=> atom(V1) | cocon_eq(V1,dt(bool,BV1)),cocon_and(dt(bool,BV1),V2,VJ).
attrjoin(symmetric,V1,V2,VJ)
    <=> atom(V2) | cocon_eq(V2,dt(bool,BV2)),cocon_and(V1,dt(bool,BV2),VJ).
attrjoin(symmetric,dt(bool,V1),dt(bool,V2),DVJ)
    <=> cocon_and(dt(bool,V1),dt(bool,V2),DVJ).
attrmeet(symmetric,V1,V2,VJ)
    <=> cocon_or(V1,V2,VJ).

ac(E1,attr(symmetric,B1)), attrule(symmetric,negate,[E,E1]) ==>
    gensym(coco_,Dim) |
    ec(E,t('Array',[tt(t('Top',[],[])),dt(int,2),
    dt(cocontpl(2,interval(int)),[Dim,Dim])],[])),
    ac(E,attr(symmetric,B1)).
attrcon(E,attr(symmetric,B)),attrule(symmetric,negate,[E,E1]) ==>
    gensym(coco_,Dim) |
    ec(E1,t('Array',[tt(t('Top',[],[])),dt(int,2),
          dt(cocontpl(2,interval(int)),[Dim,Dim])],[attr(symmetric,B)])),
```

```
    attrcon(E1,attr(symmetric,B)).

ac(E1,attr(symmetric,B1)),
    ac(E2,attr(symmetric,B2)),
    attrule(symmetric,plus,[E,E1,E2]) ==>
        (B1==dt(bool,1)),(B2==dt(bool,1)), gensym(coco_,Dim) |
        ec(E,t('Array',[tt(t('Top',[],[])),dt(int,2),
                     dt(cocontpl(2,interval(int)),[Dim,Dim])],[])),
        ac(E,attr(symmetric,dt(bool,1))).
attrcon(E,attr(symmetric,dt(bool,1))),
    attrule(symmetric,plus,[E,E1,E2]) ==>
        gensym(coco_,Dim) |
        ec(E1,t('Array',[tt(t('Top',[],[])),dt(int,2),
                        dt(cocontpl(2,interval(int)),[Dim,Dim])],
            [attr(symmetric,dt(bool,1))])),
        attrcon(E1,attr(symmetric,dt(bool,1))),
        attrcon(E2,attr(symmetric,dt(bool,1))),
        ec(E2,t('Array',[tt(t('Top',[],[])),dt(int,2),
                       dt(cocontpl(2,interval(int)),[Dim,Dim])],
           [attr(symmetric,dt(bool,1))])).

ac(E2,attr(symmetric,B2)),
attrule(symmetric,scalarmult,[E,E1,E2]) ==>
    gensym(coco_,Dim) |
    ec(E,t('Array',[tt(t('Top',[],[])),dt(int,2),
                          dt(cocontpl(2,interval(int)),[Dim,Dim])],[])),
    ac(E,attr(symmetric,B2)).
attrcon(E,attr(symmetric,B)),
attrule(symmetric,scalarmult,[E,E1,E2]) ==>
    gensym(coco_,Dim) |
    ec(E2,t('Array',[tt(t('Top',[],[])),dt(int,2),
               dt(cocontpl(2,interval(int)),[Dim,Dim])],
```

```
        [attr(symmetric,B)])),
    attrcon(E2,attr(symmetric,B)).

/*
```

6.7.2 Scalar Range

Scalar variables can be given a range, the value of the variable is contained in that range.

The join is the union of two ranges, meet is the intersection, subattribute relation is the contained relation on intervals. Addition, substraction, multiplication of scalar variables has the anticipated range results. Division is tedious because the presence of 0 in the possible values for a divisor creates problems; this could probably be addressed successfully, but was not a significant concern at this time.

The range attribute is very significant in itself in that any concrete program requires that all instantiated terms must be of a size representable by the underlying system (this might include supporte for arbitrary size numbers, but generally a single register implementation is desirable). By providing the range of scalar variables, evidence of this can be provided, or problem areas highlighted.

There is an issue with the current implementation in that the type system is only built to manage 4 byte floating point and integer number representations. This means that the range is expressible in two 4 byte numbers, is symbolic and uncomputable by the type system, or has not been provided at all so no estimate can be made. This problem has been left for future work to address.

Note that it is possible that unexpressible terms will be eliminated by other layers of the Coconut system.

```
 */
attrjoin('Range',dt(interval(T),V1),dt(interval(T),V2),DVJ) <=>
    atom(V1) |
      cocon_eq(dt(interval(T),V1),dt(interval(T),(V1L,V1U))),
      attrjoin('Range',dt(interval(T),(V1L,V1U)),dt(interval(T),V2),DVJ).
attrjoin('Range',dt(interval(T),V1),dt(interval(T),V2),DVJ) <=>
```

```
    atom(V2) |
      cocon_eq(dt(interval(T),V2),dt(interval(T),(V2L,V2U))),
      attrjoin('Range',dt(interval(T),V1),dt(interval(T),(V2L,V2U)),DVJ).
attrjoin('Range',dt(interval(T),(V1L,V1U)),dt(interval(T),(V2L,V2U)),DVJ) <=>
    min(V1L,V2L,VJL),max(V1U,V2U,VJU),DVJ=dt(interval(T),(VJL,VJU)).

attrmeet('Range',dt(interval(T),V1),dt(interval(T),V2),DVJ) <=>
    atom(V1) |
        cocon_eq(dt(interval(T),V1),dt(interval(T),(V1L,V1U))),
        attrmeet('Range',dt(interval(T),(V1L,V1U)),dt(interval(T),V2),DVJ).
attrmeet('Range',dt(interval(T),V1),dt(interval(T),V2),DVJ) <=>
    atom(V2) |
      cocon_eq(dt(interval(T),V2),dt(interval(T),(V2L,V2U))),
      attrmeet('Range',dt(interval(T),V1),dt(interval(T),(V2L,V2U)),DVJ).
attrmeet('Range',dt(interval(T),(V1L,V1U)),dt(interval(T),(V2L,V2U)),DVJ) <=>
    max(V1L,V2L,VJL),min(V1U,V2U,VJU),DVJ=dt(interval(T),(VJL,VJU)).


attrcon(E,attr('Range',dt(interval(T),(L,U))))
  <=> (U < L) | attrcon(E,attr('Range',dt(null,[]))),
                error(E,attr('Range',dt(interval(T),(L,U)))).

/*

 */
attrcon(E,attr('Range',dt(interval(T),(L,U)))),
  attrule('Range',negate,[E,E1])
     ==> (number(L)-> U1 is (- L)),(number(U)-> L1 is (-U)) |
             ec(E1,t('Scalar',[])),
             attrcon(E1,attr('Range',dt(interval(T),(L1,U1)))).
ac(E1,attr('Range',dt(interval(T),(L1,U1)))),
  attrule('Range',negate,[E,E1])
     ==>  (number(L1)-> U is (- L1)),(number(U1)-> L is (-U1)) |
```

```
              ec(E,t('Scalar',[])), ac(E,attr('Range',dt(interval(T),(L,U)))).

attrcon(E,attr('Range',dt(interval(T),(L,U)))),
    ac(E1,attr('Range',dt(interval(T1),(L1,U1)))),
    attrule('Range',plus,[E,E1,E2]) ==> intfloat(T,T1,T2) |
        add(L1,L2,L),add(U1,U2,U),
        attrcon(E2,attr('Range',dt(interval(T2),(L2,U2)))).
attrcon(E,attr('Range',dt(interval(T),(L,U)))),
    ac(E2,attr('Range',dt(interval(T2),(L2,U2)))),
    attrule('Range',plus,[E,E1,E2]) ==> intfloat(T,T2,T1) |
        add(L1,L2,L),add(U1,U2,U),
        attrcon(E1,attr('Range',dt(interval(T),(L1,U1)))).
ac(E1,attr('Range',dt(interval(T1),(L1,U1)))),
    ac(E2,attr('Range',dt(interval(T2),(L2,U2)))),
    attrule('Range',plus,[E,E1,E2]) ==> intfloat(T1,T2,T) |
        add(L1,L2,L),add(U1,U2,U),
        ac(E,attr('Range',dt(interval(T),(L,U)))).

intfloat(T1,T2,T) :- (T1==float ; T2==float)->(T=float);(T=int).

/*
```

6.8 CHR Arithmetic

CHRs can provide a rudimentary form of symbolic arithmetic. This has been used to resolve data terms in attribute class rules, and subattribute, join and meet relations.

The positive side to the CHR approach is that symbolic constraints are left behind when they cannot be resolved, and these are often quite useful in themselves. For example, having the constraint "coco_le(pointSize, maxPtCap)" as part of the type judgement makes an excellent addition to the proof obligations for the module.

The negative side to the CHR algebra is that when a constraint cannot be satisfied, the error cannot be traced back (easily) to the rule that spawned the constraint

originally. If the above constraint were to be shown false, the system can either fail or record an error and proceed, but cannot roll back the rule that spawned the constraint.

This restriction has forced a very conservative approach to generating the attribute class rules. It also forced the inference of dependent type rules to be computed in a Prolog based context manager, which could make these restraints predicates and block inappropriate actions (more in 6.2.4).

6.8.1 Boolean Algebra

Constraint Boolean algebra, wrapped with HUSC data term type, adapted from Boolean Constraints, Thom Fruehwirth ECRC 1991-1993, `http://chr.informatik.uni-ulm.de/~webchr`

```
 */
cocon_and(dt(bool,0),_,Z) <=> Z=dt(bool,0).
cocon_and(_,dt(bool,0),Z) <=> Z=dt(bool,0).
cocon_and(dt(bool,1),dt(bool,1),Z) <=> Z=dt(bool,1).
cocon_and(X,Y,dt(bool,1)) <=> X=dt(bool,1),Y=dt(bool,1).
cocon_and(dt(bool,1),Y,dt(bool,0)) <=> Y=dt(bool,0).
cocon_and(X,dt(bool,1),dt(bool,0)) <=> X=dt(bool,0).
cocon_and(X,Y,Z1) \ cocon_and(X,Y,Z2) <=> Z2=Z1.
cocon_and(X,Y,Z1) \ cocon_and(Y,X,Z2) <=> Z2=Z1.

cocon_or(dt(bool,1),_,Z) <=> Z=dt(bool,1).
cocon_or(_,dt(bool,1),Z) <=> Z=dt(bool,1).
cocon_or(dt(bool,0),dt(bool,0),Z) <=> Z=dt(bool,0).
cocon_or(X,Y,dt(bool,0)) <=> X=dt(bool,0),Y=dt(bool,0).
cocon_or(dt(bool,0),Y,dt(bool,1)) <=> Y=dt(bool,1).
cocon_or(X,dt(bool,1),dt(bool,1)) <=> X=dt(bool,1).
cocon_or(X,Y,Z1) \ cocon_or(X,Y,Z2) <=> Z2=Z1.
cocon_or(X,Y,Z1) \ cocon_or(Y,X,Z2) <=> Z2=Z1.
```

```
cocon_xor(dt(bool,0),X,Y) <=> X=Y.
cocon_xor(X,dt(bool,0),Y) <=> X=Y.
cocon_xor(X,Y,dt(bool,0)) <=> X=Y.
cocon_xor(dt(bool,1),X,Y) <=> neg(X,Y).
cocon_xor(X,dt(bool,1),Y) <=> neg(X,Y).
cocon_xor(X,Y,dt(bool,1)) <=> neg(X,Y).
cocon_xor(X,X,Y) <=> Y=dt(bool,0).
cocon_xor(X,Y,X) <=> Y=dt(bool,0).
cocon_xor(Y,X,X) <=> Y=dt(bool,0).
cocon_xor(X,Y,A) \ cocon_xor(X,Y,B) <=> A=B.
cocon_xor(X,Y,A) \ cocon_xor(Y,X,B) <=> A=B.

cocon_not(dt(bool,0),X) <=> X=dt(bool,1).
cocon_not(X,dt(bool,0)) <=> X=dt(bool,1).
cocon_not(dt(bool,1),X) <=> X=dt(bool,0).
cocon_not(X,dt(bool,1)) <=> X=dt(bool,0).
cocon_not(X,X) <=> fail.
cocon_not(X,Y) \ cocon_not(Y,Z) <=> X=Z.
cocon_not(X,Y) \ cocon_not(Z,Y) <=> X=Z.
cocon_not(Y,X) \ cocon_not(Y,Z) <=> X=Z.
%% Interaction with other boolean constraints
cocon_not(X,Y) \ cocon_and(X,Y,Z) <=> Z=dt(bool,0).
cocon_not(Y,X) \ cocon_and(X,Y,Z) <=> Z=dt(bool,0).
cocon_not(X,Z) , cocon_and(X,Y,Z) <=> X=dt(bool,1),Y=dt(bool,0),Z=dt(bool,0).
cocon_not(Z,X) , cocon_and(X,Y,Z) <=> X=dt(bool,1),Y=dt(bool,0),Z=dt(bool,0).
cocon_not(Y,Z) , cocon_and(X,Y,Z) <=> X=dt(bool,0),Y=dt(bool,1),Z=dt(bool,0).
cocon_not(Z,Y) , cocon_and(X,Y,Z) <=> X=dt(bool,0),Y=dt(bool,1),Z=dt(bool,0).
cocon_not(X,Y) \ cocon_or(X,Y,Z) <=> Z=dt(bool,1).
cocon_not(Y,X) \ cocon_or(X,Y,Z) <=> Z=dt(bool,1).
cocon_not(X,Z) , cocon_or(X,Y,Z) <=> X=dt(bool,0),Y=dt(bool,1),Z=dt(bool,1).
cocon_not(Z,X) , cocon_or(X,Y,Z) <=> X=dt(bool,0),Y=dt(bool,1),Z=dt(bool,1).
cocon_not(Y,Z) , cocon_or(X,Y,Z) <=> X=dt(bool,1),Y=dt(bool,0),Z=dt(bool,1).
```

```
cocon_not(Z,Y) , cocon_or(X,Y,Z) <=> X=dt(bool,1),Y=dt(bool,0),Z=dt(bool,1).
cocon_not(X,Y) \ cocon_xor(X,Y,Z) <=> Z=dt(bool,1).
cocon_not(Y,X) \ cocon_xor(X,Y,Z) <=> Z=dt(bool,1).
cocon_not(X,Z) \ cocon_xor(X,Y,Z) <=> Y=dt(bool,1).
cocon_not(Z,X) \ cocon_xor(X,Y,Z) <=> Y=dt(bool,1).
cocon_not(Y,Z) \ cocon_xor(X,Y,Z) <=> X=dt(bool,1).
cocon_not(Z,Y) \ cocon_xor(X,Y,Z) <=> X=dt(bool,1).
cocon_not(X,Y) , cocon_imp(X,Y) <=> X=dt(bool,0),Y=dt(bool,1).
cocon_not(Y,X) , cocon_imp(X,Y) <=> X=dt(bool,0),Y=dt(bool,1).

cocon_imp(dt(bool,0),_) <=> true.
cocon_imp(X,dt(bool,0)) <=> X=dt(bool,0).
cocon_imp(dt(bool,1),X) <=> X=dt(bool,1).
cocon_imp(_,dt(bool,1)) <=> true.
cocon_imp(X,X) <=> true.
cocon_imp(X,Y),cocon_imp(Y,X) <=> X=Y.

/*

 */
/*
 * Float and Integer operations,
 * add (can't use plus, builtin to prolog), negate.
*/
add(X,Y,Z) <=> number(X),number(Y),number(Z) | Z =:= (X+Y).
add(X,Y,Z) <=> number(X),number(Y) | Z is (X+Y).
add(X,Y,Z) <=> number(X),number(Z) | Y is (Z-X).
add(X,Y,Z) <=> number(Y),number(Z) | X is (Z-Y).
add(X,0,Z) <=> Z=X.
add(0,Y,Z) <=> Z=Y.
add(X,Y,Z),neg(X,Y) <=> Z=0.
add(X,Y,Z),neg(Y,X) <=> Z=0.
add(X,Y,0) <=> neg(X,Y).
```

```
add(X,Y,Z1) \ add(X,Y,Z2) <=> Z1=Z2.
add(X,Y,Z1) \ add(Y,X,Z2) <=> Z1=Z2.

neg(X,Y) <=> number(X),number(Y) | X == (-Y).
neg(X,Y) <=> number(X) | Y=(-X).
neg(X,Y) <=> number(Y) | X=(-Y).
neg(X,Y),neg(Y,X) <=> X=Y.
neg(X,X) <=> fail.
neg(X,Y) \ neg(Z,Y) <=> Z=X.
neg(X,Y) \ neg(X,Z) <=> Z=Y.

/*
```

Basic operators for unwrapped numbers, i.e. integers and floats. From CHR solver "INEQUALITIES with MINIMIUM and MAXIMUM on Terms", Thom Fruehwirth ECRC 1991-1993, http://chr.informatik.uni-ulm.de/ webchr

```
 */
X ~= X <=> fail.
X ~= Y <=> ground(X),ground(Y) | X\==Y.

X geq Y :- Y leq X.
X grt Y :- Y lss X.

/* leq */
built_in     @ X leq Y <=> ground(X),ground(Y) | X @=< Y.
reflexivity  @ X leq X <=> true.
antisymmetry @ X leq Y, Y leq X <=> X = Y.
transitivity @ X leq Y, Y leq Z ==> X \== Y, Y \== Z, X \== Z | X leq Z.
subsumption  @ X leq N \ X leq M <=> N@<M | true.
subsumption  @ M leq X \ N leq X <=> N@<M | true.

/* lss */
```

```
built_in     @ X lss Y <=> ground(X),ground(Y) | X @< Y.
irreflexivity@ X lss X <=> fail.
transitivity @ X lss Y, Y lss Z ==> X \== Y, Y \== Z | X lss Z.
transitivity @ X leq Y, Y lss Z ==> X \== Y, Y \== Z | X lss Z.
transitivity @ X lss Y, Y leq Z ==> X \== Y, Y \== Z | X lss Z.
subsumption  @ X lss Y \ X leq Y <=> true.
subsumption  @ X lss N \ X lss M <=> N@<M | true.
subsumption  @ M lss X \ N lss X <=> N@<M | true.
subsumption  @ X leq N \ X lss M <=> N@<M | true.
subsumption  @ M leq X \ N lss X <=> N@<M | true.
subsumption  @ X lss N \ X leq M <=> N@<M | true.
subsumption  @ M lss X \ N leq X <=> N@<M | true.

/* neq */
built_in     @ X neq Y <=> X ~= Y | true.
irreflexivity@ X neq X <=> fail.
subsumption  @ X neq Y \ Y neq X <=> true.
subsumption  @ X lss Y \ X neq Y <=> true.
subsumption  @ X lss Y \ Y neq X <=> true.
simplification @ X neq Y, X leq Y <=> X lss Y.
simplification @ Y neq X, X leq Y <=> X lss Y.

/* MIN */
labeling, min(X, Y, Z)#Pc <=>
( X leq Y, Z = X ; Y lss X, Z = Y),
labeling
pragma passive(Pc).

built_in @ min(X, Y, Z) <=> ground(X),ground(Y) | (X@=<Y -> Z=X ; Z=Y).
built_in @ min(X, Y, Z) <=> Z~=X | Z = Y, Y lss X.
built_in @ min(Y, X, Z) <=> Z~=X | Z = Y, Y lss X.
min_eq @ min(X, X, Y) <=> X = Y.
```

```
min_leq @ Y leq X \ min(X, Y, Z) <=> Y=Z.
min_leq @ X leq Y \ min(X, Y, Z) <=> X=Z.
min_lss @ Z lss X \ min(X, Y, Z) <=> Y=Z.
min_lss @ Z lss Y \ min(X, Y, Z) <=> X=Z.
functional @ min(X, Y, Z) \ min(X, Y, Z1) <=> Z1=Z.
functional @ min(X, Y, Z) \ min(Y, X, Z1) <=> Z1=Z.
propagation @ min(X, Y, Z) ==> X\==Y | Z leq X, Z leq Y.

/* MAX */
labeling, max(X, Y, Z)#Pc <=>
( X leq Y, Z = Y ; Y lss X, Z = X),
labeling
pragma passive(Pc).
built_in @ max(X, Y, Z) <=> ground(X),ground(Y) | (Y@=<X -> Z=X ; Z=Y).
built_in @ max(X, Y, Z) <=> Z~=X | Z = Y, X lss Y.
built_in @ max(Y, X, Z) <=> Z~=X | Z = Y, X lss Y.
max_eq @ max(X, X, Y) <=> X = Y.
max_leq @ Y leq X \ max(X, Y, Z) <=> X=Z.
max_leq @ X leq Y \ max(X, Y, Z) <=> Y=Z.
max_lss @ X lss Z \ max(X, Y, Z) <=> Y=Z.
max_lss @ Y lss Z \ max(X, Y, Z) <=> X=Z.
functional @ max(X, Y, Z) \ max(X, Y, Z1) <=> Z1=Z.
functional @ max(X, Y, Z) \ max(Y, X, Z1) <=> Z1=Z.
propagation @ max(X, Y, Z) ==> X\==Y | X leq Z, Y leq Z.

/*
```

*/
/*

6.9 Named and Generic Subtypes, Joins and Meets

The subtype, join and meet relations for HUSC generic shapes and named types hierarchy (not including attributes) are in this module. They are implemented as Prolog predicates, instead of CHRs.

The dependent variable context is carried throughout the clauses presented here, with the input context and output context from each judgement being the two lists at the end of the clause arguments, respectively.

```
 */
:- module( subtype, [join/3, join/5, meet/3, meet/5, st/2, st/4]).
:- use_module(context).
:- use_module(attribute).
/*
```

Subtype and Stated Subtype Relations

The subtype relation is represented by predicate st/2, st/4, where the 4 argument case unifies an input and output local context. The 2 argument case is processed as with an empty context as input, then throws the context information away when done.

Also note that for consistency, the two element type description is translated into a three element type description with an empty attribute list.

The following rules are implemented, in this order, answering $S \preceq T$:

1. subtype is reflexive
2. bottom ($\perp_T$) is always a subtype
3. everything is a subtype of top ($\top_T$)
4. if the type names are the same, the dependent parameters of S are subterms of T (data terms equal, type terms subtypes) and all attributes of T are also represented in S

5. if T is a type variable, and has no current restrictions, T is set to S in the context and the subtype relation holds.

6. If T is a type variable, and is constrained to be the subtype of T*, then test $T = S \sqcup T*$ in the context; if this does not break any subtype constraints on T, that is the new value of T, otherwise return the value $S \preceq T*$.

7. If S is a type variable, and is constrained to be the subtype of S*, then test $S = S * \sqcup T$ in the context; if this does not break any supertype constraints on S, that is the new value of S, otherwise return the value $S* \preceq T$.

8. if the types do not share the same name, then take the stated supertype (sst) of S, with the dependent arguments established for S, to get S*, then return $S* \preceq T$.

9. Top ($\top_T$) is never a subtype so return False if this is the only choice (note that reflexivity is taken care of already so $\top_T \preceq \top_T$).

The stated subtype relation is the explicit subtype relation defined in the type definitions. Note that the search up the subtype hierarchy is always terminated by top.

```
 */
st(S,T) :- st(S,T,_,[]).
st(S,S,CTX,CTX).
st(t(S,SAs),T,CTXOut,CTXIn) :- st(t(S,SAs,[]),T,CTXOut,CTXIn).
st(S,t(T,TAs),CTXOut,CTXIn) :- st(S,t(T,TAs,[]),CTXOut,CTXIn).
st(t('Bottom',[],[]),_,CTX,CTX).
st(t('Top',[],[]),_,CTX,CTX) :- !, fail.
st(t(T,SArgs,SAttrs), t(T,TArgs,TAttrs), CTXOut, CTXIn) :-
    matchlist(SArgs,TArgs,CTXA,CTXIn),
    subattrs(SAttrs,TAttrs,CTXOut,CTXA).
st(t(T,TArgs,TAttrs), tv(V,VSup), CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,tv(V,VSup),VC),
      ctxeval(CTXIn,t(T,TArgs,TAttrs),T2),
```

```
      ( (VC==tv(V,VSup)->ctxupd(supt,VC,T2,CTXOut,CTXIn))
      ; !,st(T2,VC,CTXOut,CTXIn) ) );
    !,fail.
st(tv(V,VSup), t(T,TArgs,TAttrs), CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,tv(V,VSup),VC),
      ctxeval(CTXIn,t(T,TArgs,TAttrs),T2),
      ( (VC==tv(V,VSup)->ctxupd(subt,VC,T2,CTXOut,CTXIn))
      ; !,st(VC,T2,CTXOut,CTXIn) ) );
    !,fail.
st(tv(SV,SVSup), tv(TV,TVSup), CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,tv(SV,SVSup),SVC),
      ctxeval(CTXIn,tv(TV,TVSup),TVC),
      ( ((SVC==tv(SV,SVSup),TVC==tv(TV,TVSup))
          ->ctxupd(subt,SVC,TVC,CTXOut,CTXIn))
      ; !,st(SVC,TVC,CTXOut,CTXIn) ) );
    !,fail.
st(t(S,SAs,SAttrs), t(T,TAs,TAttrs), CTXOut, CTXIn) :-
    sst(t(S,SAs),t(U,UAs),CTXA,CTXIn)
        -> !,st(t(U,UAs,SAttrs),t(T,TAs,TAttrs),CTXOut,CTXA).
/*
```

Stated SubType Relation

The stated (defined) subtype relation returns the explicitly defined subtype from the type definition. It uses matchlist from context.pl to unify the dependent variables in the subtype term with those of the supertype term, with the type top relationship hardcoded in the predicate rules.

The last argument of the predicate for the type definition (type) is st for a subtype relationship, and bt for a variant (based on) relationship; obviously only subtype relationships are explored.

Note that the third argument of type is not used, and is always an empty list. This should be removed at some point.

```
 */
```

```
sst(t(X,XAs,XAttrs),t(Y,YAs,XAttrs),CTXOut,CTXIn) :-
    sst(t(X,XAs),t(Y,YAs),CTXOut,CTXIn).
sst(t(X,XAs),t(Y,YAs),CTXOut,CTXIn) :-
    type(t(X,XPs,_), t(Y,YAs,_), _, st),
    matchlist(XAs,XPs,CTXOut,CTXIn).
sst(t(X,XAs,_),t('Top',[],[]),CTXOut,CTXIn) :-
    type(t(X,XPs,[]),_,_,_),
    matchlist(XAs,XPs,CTXOut,CTXIn).
/*
```

6.9.1 Generic Join Relation

The join(type1,type2,type3) sets type3 to the minimal supertype, or least general generalization, of types 1 and 2, with all types in the internal type representation (t).

The following rules apply to joins, with the list being applied top to bottom until a result is obtained:

1. The join of two identical types is the type itself, for type terms and type variables.

2. The join of two types of the same name is the name with the first parameter joined and the meet of the remaining parameters; the number of parameters must be the same.

3. If one argument is the subtype of the other, then the supertype is the join value.

4. Failing this test, the supertype of one of the terms is computed, and the result is that joined with the other term.

5. the join of any item and the top type is top, the ambiguous type judgement.

The meet of data terms is equality of those terms (i.e. they must be identical), and the meet of type terms is the least subtype of the two, i.e. the regular definition of meet. The number of parameters must be the same in both t representations for the join to be successful.

```
 */
join(S1,S2,T) :- join(S1,S2,T,_,[]).
join(t(T,TAs),t(T,TAs),t(T,TAs),CTX,CTX).
join(t(T,TAs,TAttrs),t(T,TAs,TAttrs),t(T,TAs,TAttrs),CTX,CTX).
join(tv(V,VSup),tv(V,VSup),tv(V,VSup),CTX,CTX).
join(t(T,[tt(RA) | As],TAAs), t(T,[tt(RB) | Bs],TBAs),
       t(T, [tt(RJ) | Js],TJAs),CTXOut,CTXIn) :-
          join(RA,RB,RJ,CTXA,CTXIn),
          joinattrs(TAAs,TBAs,TJAs,CTXB,CTXA),
          meetterm(As, Bs, Js, CTXOut,CTXB),!.
join(T1,T2,T3,CTXOut,CTXIn) :-
    st(T1,T2,CTXOut,CTXIn),ctxeval(CTXOut,T2,T3).
join(T1,T2,T3,CTXOut,CTXIn) :-
    st(T2,T1,CTXOut,CTXIn),ctxeval(CTXOut,T1,T3).
join(T1,T2,T3B,CTXOut,CTXIn) :-
    sst(T1,U,CTXA,CTXIn) -> join(U,T2,T3,CTXOut,CTXA),ctxeval(CTXOut,T3,T3B).
join(_,_,t('Top',[],[]),CTX,CTX).
/*
```

6.9.2 Generic Meet Relation

This is similar to the join explained above, except that if neither type is a subtype of the other, the meet is bottom.

Note that the dependent arguments of the meet are the join of the dependent arguments of the operands. This means that if an expression is constrained to be two similar shapes with type parameters, say a function for convenience, then the function that satisfies both constraints must accept arguments from either parameter type, i.e. is covariant in its parameters and contravariant in its result.

```
 */
meet(T1,T2,S) :- meet(T1,T2,S,_,[]).
meet(t(T,TAs),t(T,TAs),t(T,TAs),CTX,CTX).
meet(t(T,TAs,TAttrs),t(T,TAs,TAttrs),t(T,TAs,TAttrs),CTX,CTX).
```

```
meet(t(T,[tt(RA) | As]), t(T,[tt(RB) | Bs]), t(T, [tt(RJ) | Js]), CTXOut,CTXIn) :-
    meet(RA,RB,RJ,CTXA,CTXIn),jointerm(As, Bs, Js, CTXOut, CTXA),!.
meet(t(T,[tt(RA) | As],TAAs), t(T,[tt(RB) | Bs],TBAs),
         t(T, [tt(RJ) | Js],TJAs), CTXOut,CTXIn) :-
             meet(RA,RB,RJ,CTXA,CTXIn),
             meetattrs(TAAs,TBAs,TJAs,CTXB,CTXA),
             jointerm(As, Bs, Js, CTXOut, CTXB),!.
meet(T1,T2,T3, CTXOut,CTXIn) :-
    st(T1,T2, CTXOut,CTXIn),ctxeval(CTXOut,T1,T3).
meet(T1,T2,T3, CTXOut,CTXIn) :-
    st(T2,T1, CTXOut,CTXIn),ctxeval(CTXOut,T2,T3).
meet(_,_,t('Bottom',[]),CTX,CTX).
/*
```

6.9.3 Joins / Meets over Internal Terms

This performs the mechanics of performing a meet or a join over a list of dependent variables. The internal representation of terms must be compared predicate by predicate. In particular, the internal data structure cocontpl is used for checking that a fixed size list of internal terms is of the correct form. This allowed a generic definition of a function and array to be created and the representation checked.

The most important aspect of these mechanics is the management of the contexts. The input and output context lists are unified through the ctxeval predicate from context.pl. This makes sure each variable term (variables are stored as Prolog atoms, not Prolog variables) is evaluated relative to other constraints generated during the processing of the list of joins / meets.

```
 */
jointerm([],[],[],CTX,CTX).
jointerm(As,tt(cocontpl(N,T)),Js,CTXOut,CTXIn) :-
    ttscheck(As,N,T,CTXOut,CTXIn),!,ctxeval(CTXOut,As,Js).
jointerm(As,dt(cocontpl(N,T)),Js,CTX,CTX) :-
    dtscheck(As,N,T),!,ctxeval(CTX,As,Js).
```

```
jointerm(tt(cocontpl(N,T)),Bs,Js,CTXOut,CTXIn) :-
    ttscheck(Bs,N,T,CTXOut,CTXIn),!,ctxeval(CTXOut,Bs,Js).
jointerm(dt(cocontpl(N,T)),Bs,Js,CTX,CTX) :-
    dtscheck(Bs,N,T),!,ctxeval(CTX,Bs,Js).
jointerm([tt(t(A,As)) | MAs],[tt(t(B,Bs)) | MBs],
              [tt(t(J,Js)) | MJs],CTXOut,CTXIn) :-
                 join(t(A,As),t(B,Bs),t(J,Js),CTXA,CTXIn),!,
                 jointerm(MAs,MBs,MJs,CTXOut,CTXA).
jointerm([tt(t(A,As,AAttrs)) | MAs],[tt(t(B,Bs,BAttrs)) | MBs],
             [tt(t(J,Js,JAttrs)) | MJs],CTXOut,CTXIn) :-
                join(t(A,As,AAttrs),t(B,Bs,BAttrs),t(J,Js,JAttrs),CTXA,CTXIn),!,
                jointerm(MAs,MBs,MJs,CTXOut,CTXA).
jointerm([dt(T,V1)|MAs],[dt(T,V2) | MBs],
             [dt(T,V1) | MCs],CTXOut,CTXIn) :-
                 match(T,V1,V2,CTXA,CTXIn),!,
                 jointerm(MAs,MBs,MCs,CTXOut,CTXA).

meetterm([],[],[],CTX,CTX).
meetterm(As,tt(cocontpl(N,T)),Js,CTXOut,CTXIn) :-
    ttscheck(As,N,T,CTXOut,CTXIn),!,ctxeval(CTXOut,As,Js).
meetterm(As,dt(cocontpl(N,T)),Js,CTX,CTX) :-
    dtscheck(As,N,T),!,ctxeval(CTX,As,Js).
meetterm(tt(cocontpl(N,T)),Bs,Js,CTXOut,CTXIn) :-
    ttscheck(Bs,N,T,CTXOut,CTXIn),!,ctxeval(CTXOut,Bs,Js).
meetterm(dt(cocontpl(N,T)),Bs,Js,CTX,CTX) :-
    dtscheck(Bs,N,T),!,ctxeval(CTX,Bs,Js).
meetterm([tt(t(A,As)) | MAs],[tt(t(B,Bs)) | MBs],
              [tt(t(J,Js)) | MJs],CTXOut,CTXIn) :-
                 meet(t(A,As),t(B,Bs),t(J,Js),CTXA,CTXIn),
                 meetterm(MAs,MBs,MJs,CTXOut,CTXA).
meetterm([tt(t(A,As,AAttrs)) | MAs],[tt(t(B,Bs,BAttrs)) | MBs],
              [tt(t(J,Js,JAttrs)) | MJs],CTXOut,CTXIn) :-
```

```
                 meet(t(A,As,AAttrs),t(B,Bs,BAttrs),t(J,Js,JAttrs),CTXA,CTXIn),
                 meetterm(MAs,MBs,MJs,CTXOut,CTXA).
meetterm([dt(T,V1)|MAs],[dt(T,V2) | MBs],
               [dt(T,V1) | MCs],CTXOut,CTXIn) :-
                  match(T,V1,V2,CTXA,CTXIn),!,
                  meetterm(MAs,MBs,MCs,CTXOut,CTXA).
/*
```

6.9.4 Testing Types against their Definition

Each declared type term must be a valid instance of the type definition. This is tested by validtype, using matchlist to ensure the dependent arguments are appropriate for the dependent parameters in the internal representation terms.

It wasn't clear what to do with the errors, so the flag variable Str is set to valid or invalid.

```
 */
validtype( t('Top',_,[]), Str) :- Str=valid.
validtype( t(T,DTArgs), Str) :- validtype(t(T,DTArgs,[]),Str).
validtype( t(T,DTArgs,TAttrs), Str) :-
     type( t(T,DTparms,TMAs), _, _, _ ),
     matchlist(DTArgs, DTparms, CTX, []),
     validattrs( TAttrs, t(T,DTArgs,TAttrs), CTXA, CTX, Str ),
     subattrs( TAttrs,TMAs,_,CTXA ),
     Str=valid.
validtype( _, Str) :- Str=invalid.
/*
```

*/

/*

6.10 Attribute Class Part II

This file contains the Prolog subattribute predicate and the rule pattern matching predicate for the attribute class definitions. It was originally specified to be generated from the HUSC source for the attribute classes, but this file was hand coded. This version contains the range attribute for scalars and symmetric attribute for arrays.

This file was intended to hold the attribute CHRs too, but as noted in constraint.pl, the CHRs must all be in the same module, which makes generating the file more difficult.

The internal representation of attributs is the attr(class,value) predicate. The class is the class name as an atom. The value is an internal data term (dt), of type int, float or boolean.

```
 */
:- module(attribute,[reg_attrule/2,subattr/5,subattrs/4,subattrs/5,setattr/3]).
:- use_module(subtype).
:- use_module(context).
/*
```

6.10.1 Subattribute List

When testing $S \preceq T$, all of the attributes of T must appear in S as subattributes. The subattrs predicate applies the class specific subattribute relation to each of the attributes in the two argument types, but within a consistent local term context (6.2.4).

The main point of subattrs is to identify all of the attribute classes that play a role in either type, then test each subattribute relation in turn. The mechanics are handled by

maplist pulling out the attribute classes from the type descriptions (t)

merge_set merging the two lists

valattr get the values from the matching attributes of each class

subattr invoke the class specific subattribute predicate

There are two subattribute rules that are common for all classes.

1. no attribute is a subattribute of a null valued attribute, representing a $\perp$ evaluation of the property, resulting from a failure to unify the subattribute relations in the context.

2. subattribute is reflexive

3. a null valued attribute is always a subattribute of a non-null valued attribute

```
 */
subattrs(t(_,_,SAttrs),t(_,_,TAttrs),CTXOut,CTXIn) :-
    maplist(getac,SAttrs,SACs),maplist(getac,TAttrs,TACs),
    merge_set(SACs,TACs,ACs),subattrs(ACs,SAttrs,TAttrs,CTXOut,CTXIn).
subattrs(Ss,Ts,CTXOut,CTXIn) :-
    maplist(getac,Ss,SACs),maplist(getac,Ts,TACs),
    merge_set(SACs,TACs,ACs),subattrs(ACs,Ss,Ts,CTXOut,CTXIn).
subattrs([],_,_,CTX,CTX).
subattrs([AC|ACs],S,T,CTXOut,CTXIn) :-
    ( valattr(AC,T,TV) ->
      ( valattr(AC,S,SV) ->
        ( subattr(AC,SV,TV,CTXA,CTXIn) )
      ; ( !,fail ) )
    ; (CTXA = CTXIn) ),
    subattrs(ACs,S,T,CTXOut,CTXA).

getac(attr(AC,_),AC).
valattr(AC,t(_,_,As),V) :- member(attr(AC,V), As).
valattr(AC,As,V) :- member(attr(AC,V),As).  %% fails if As is not a list.
%% this must precede V <= V to block this case.
subattr(_,_,dt(null,[]),CTX,CTX) :- !,fail.
subattr(_,V,V,CTX,CTX).
subattr(_,dt(null,[]),_,CTX,CTX).
```

/*

Set Attribute in Type Description

Setattr sets the attributes of a type to includes those from the list. Setattr fails if all of the attributes of the type are the same as in the list, which breaks an endless loop in a CHR which was otherwise difficult to handle.

```
 */
setattr(t(T,TAs),Attrs,t(T,TAs,Attrs)).
setattr(t(T,TAs,TAttrs),Attrs,t(T,TAs,T2Attrs)) :-
    subset(Attrs,TAttrs)->fail
    ; setattrint(t(T,TAs,TAttrs),Attrs,t(T,TAs,T2Attrs)).

setattrint(T,[],T).
setattrint(t(T,TAs,TAttrs),[attr(AC,Val)|Attrs],
              t(T,TAs,[attr(AC,Val)|T3Attrs])) :-
                    delete(TAttrs,attr(AC,_),T2Attrs),
                    setattrint(t(T,TAs,T2Attrs),Attrs,t(T,TAs,T3Attrs)).
/*
```

6.10.2 Attribute Propagation Rule Map

These are generated relations that link an operator symbol with an (attribute,rulename) pair. This provides the list of rules to test for each expression, turned into CHR constraints by genattrule in constraint.pl.

There may be many rules for each attribute class name and operator symbol pair. The rules will only be executed if the type context matches the description and the expression pattern matches the rule. For example, the - symbol represents both unary negation and binary subtraction, but the rule will apply to the unary operator or binary operator case. The rule name doesn't have to be the same as the operator.

 */

```
reg_attrule('+',('Symmetric',plus)).
reg_attrule('+',('Range',plus)).
reg_attrule('-',('Symmetric',minus)).
reg_attrule('*',('Range',multiply)).
reg_attrule('-',('Symmetric',negate)).
reg_attrule('-',('Range',negate)).
reg_attrule('*',('Symmetric',scalarmult)).
/*
```

6.10.3 Attribute Class : General Description

There are two predicates for each class:

isattrtype tests that the attribute is applied to a value that is a subtype of the class target type.

subattr tests the subattribute relationship for the class

Attribute Class : Scalar Range

Range(T) is an interval (lb::T,ub::T) for a scalar T, where the value of a variable x with this attribute is $lb \leq x \leq ub$.

```
 */
isattrtype('Range',T,CTXOut,CTXIn) :- st(T,t(scalar,[]),CTXOut,CTXIn).
subattr('Range',dt(interval(T),V1),dt(interval(T),V2),CTX,CTX) :-
    ctxeval(CTX,V1,VC1),ctxeval(CTX,V2,VC2),
    (VC1=(VC1L,VC1U)),(VC2=(VC2L,VC2U)),
    ( (number(VC1L),number(VC2L))->(VC2L =< VC1L); VC1L==VC2L ),
    ( (number(VC1U),number(VC2U))->(VC2U >= VC1U); VC1L==VC2L ).

/*
```

Symmetric is a boolean property, either 1(true) or 0(false). Using 1,0 to keep constants separate from variables, which are implemented as atoms.

```
 */
isattrtype('Symmetric',T,CTXOut,CTXIn) :-
    gensym(coco_,Dim),
    st(T,t(array,[tt(t('Top',[])),dt(int,2),
                      dt(cocontpl(2,interval(int)),[Dim,Dim])]),CTXOut,CTXIn).

subattr('Symmetric',B1,B2) :- coco_not(B2,B2_1),coco_or(B1,B2_1,dt(bool,1)).
/*
```

Internal Arithmetic and Logic Operations

These predicates supply the logical functions that work with the internal term representations. There are more math functions in coco_math.pl, but these were all that was necessary for the time being.

```
 */
coco_not(dt(bool,0),dt(bool,1)).
coco_not(dt(bool,1),dt(bool,0)).
coco_and(dt(bool,1),dt(bool,1),dt(bool,1)).
coco_and(dt(bool,0),_,dt(bool,0)).
coco_and(_,dt(bool,0),dt(bool,0)).
coco_or(dt(bool,1),_,dt(bool,1)).
coco_or(_,dt(bool,1),dt(bool,1)).
coco_or(dt(bool,0),dt(bool,0),dt(bool,0)).

/*
```

/ /

6.11 Context Management

The main purpose of this module is to manage the local context for dependent variables and attribute properties.

Match the elements in the first list to the second to make sure they are ok. Currently ignoring attributes in type terms, need to address. Either a standard data term or an n-tuple, as a type and the number of values of that type expected, which will be satisfied by a list of any values of that type, but only as the second operator, and only as the last item of the list (last restriction is just a lazy shortcut, sorry).

```
 */
:- module(context, [minsupt/3,matchlist/2,matchlist/4,
                               ctxeval/3,match/5,ctxupd/5]).
:- use_module(subtype).

%% the two element request ignores the context
matchlist(A,B) :- matchlist(A,B,_,[]).
matchlist([],[],[],[]).
matchlist(S,S,CTX,CTX).
matchlist([tt(cocontpl(N1,T))],[tt(cocontpl(N2,T))],CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N1,N1b),ctxeval(CTXIn,N2,N2b),!,
      ( (atom(N1b),ctxupd(eq,N1b,dt(int,N2b),CTXOut,CTXIn))
      ; (atom(N2b),ctxupd(eq,N2b,dt(int,N1b),CTXOut,CTXIn))
      ; (N1b==N2b,CTXOut=CTXIn) )
    ; !,fail).
matchlist(Xs,[tt(cocontpl(N,T))],CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N,N2),!,
      ( (var(N),length(Xs,N))
      ; (atom(N),length(Xs,XL),ctxupd(eq,N,dt(int,XL),CTXA,CTXIn)) ),
      ttscheck(Xs,N2,T,CTXOut,CTXA))
    ; !,fail.
matchlist([tt(cocontpl(N,T))],Xs,CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N,N2),!,
      ( (var(N)->length(Xs,N))
        ; ( atom(N)->length(Xs,XL),ctxupd(eq,N,dt(int,XL),CTXA,CTXIn)) ),
      ttscheck(Xs,N2,T,CTXOut,CTXA))
    ; !,fail.
```

```
matchlist([dt(cocontpl(N1,T),E1)],[dt(cocontpl(N2,T),E2)],CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N1,N1b),ctxeval(CTXIn,N2,N2b),!,
      ( (atom(N1b), ctxupd(eq,N1b,dt(int,N2b),CTXA,CTXIn))
      ; (atom(N2b), ctxupd(eq,N2b,dt(int,N1b),CTXA,CTXIn))
      ; (N1b==N2b,CTXA=CTXIn) ),
      match(cocontpl(N1b,T),E1,E2,CTXOut,CTXA) )
    ; !,fail.
matchlist(Xs,[dt(cocontpl(N,T),E)],CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N,N2),ctxeval(CTXIn,Xs,XCs),!,
        dtscheck(XCs,N2,T),unwrap(XCs,XLs),!,
        match(cocontpl(N2,T),E,XLs,CTXOut,CTXIn))
    ; !,fail.
matchlist([dt(cocontpl(N,T),E)],Xs,CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N,N2),ctxeval(CTXIn,Xs,XCs),!,
        dtscheck(XCs,N2,T),unwrap(XCs,XLs),!,
        match(cocontpl(N2,T),E,XLs,CTXOut,CTXIn) )
    ; !,fail.
matchlist([tt(S) | Xs], [tt(T) | Ys],CTXOut,CTXIn) :-
    st(S,T,CTXA,CTXIn),!,matchlist(Xs,Ys,CTXOut,CTXA).
matchlist([dt(S,V1) | Xs], [dt(T,V2) | Ys], CTXOut,CTXIn) :-
    S==T, match(S,V1,V2,CTXA,CTXIn), !,
      matchlist(Xs,Ys,CTXOut,CTXA).

matchlist(_,_,_,_) :- !,fail.
%% termination case, must be last.
/*
```

match fails if the structure doesn't match and neither is an expression (i.e. atom).

```
 */
match(_,V,V,C,C). %% includes [],[]
match(list(T),[X | Xs],[Y | Ys],CO, CI) :-
    (match(list(T),Xs,Ys,CA,CI),!,match(T,X,Y,CO,CA)); (!,fail).
```

```
match(interval(T),(V1a,V1b),(V2a,V2b),CO,CI) :-
    (match(T,V1a,V2a,CA,CI),!,match(T,V1b,V2b,CO,CA));(!,fail).
match(cocontpl(N,T),E1,E2,CTXOut,CTXIn) :-
    ( ctxeval(CTXIn,N,N2),ctxeval(CTXIn,E1,E1b),ctxeval(CTXIn,E2,E2b),!,
      ( atom(E1b)->
        ( atom(E2b)->ctxupd(eq,E2b,dt(cocontpl(N2,T),E1b),CTXOut,CTXIn)
        ; ( length(E2b,N2)-> ctxupd(eq,E1b,dt(cocontpl(N2,T),E2b),CTXOut,CTXIn)
            ; !,fail) )
      ; ( atom(E2b)->
          ( length(E1b,N2)-> ctxupd(eq,E2b,dt(cocontpl(N2,T),E1b),CTXOut,CTXIn)
          ; !,fail )
        ; match(list(T),E1b,E2b,CTXOut,CTXIn) ) ) )
    ; !,fail.
match(_,A,B,C,C) :- (var(A);var(B))->A=B.
match(T,V1,V2,CO,CI) :-
    ( (number(V1),number(V2))->V1==V2,CO=CI) ;
    ( ctxeval(CI,V1,V1b),
      ctxeval(CI,V2,V2b),!,
      ( (V1b==V2b)->CO=CI
      ; ( atom(V1b)->ctxupd(eq,V1b,dt(T,V2b),CO,CI)
        ; ( atom(V2b)->ctxupd(eq,V2b,dt(T,V1b),CO,CI)
          ; ( ((V1==V1b,V2==V2b)-> !,fail)
            ; match(T,V1b,V2b,CO,CI)) ) ) ) ).

/*
```

Context update stores a context of dependent variables and constraints on proper expressions.

- ctxupd(operation, depvar or exp, depvar / exp / val, new context, old context);
- the call order from matchlist ensures if any are dependent variables,
- they will be first, and values will be last.

- dependent variables are tested when set to values, and fail if they can't be unified.

```
 */
ctxupd(OP,DE,dt(_,DE),CTX,CTX) :- member(OP,[eq,le,ge]).
ctxupd(OP,E,dt(cocontpl(N,T),DTL),[dcon(cocontpl(N,T),OP,E,L)],[]) :-
    unwrap(DTL,L).
ctxupd(OP,E,dt(T,V),[dcon(T,OP,E,V)],[]).
ctxupd(eq,E,dt(cocontpl(N,T),DTL), [dcon(cocontpl(N,T),eq,E,L) | CO], CI) :-
    delete(CI,dcon(_,eq,E,_),CA),
    (atom(DTL)->L=DTL;unwrap(DTL,L)),
    ctxrepl(E,L,CO,CA).
ctxupd(eq,E,dt(T,V), [dcon(T,eq,E,V) | CO], CI) :-
    delete(CI,dcon(_,eq,E,_),CA),
    ctxrepl(E,V,CO,CA).
%% le,lt,ge,gt operators
%% type updates, not enforcing relationships between type variables
ctxupd(subt,tv(V,VSup),t(T,TAs,TAttrs),
             [dcon(type,subt,V,t(S,SAs,SAttrs)) | CO] ,CI) :-
  ( st(VSup,t(T,TAs,TAttrs),CA,CI) ->
    ( ( member(dcon(type,subt,V,t(T2,T2As,T2Attrs)),CA)->
        ( meet(t(T,TAs,TAttrs),t(T2,T2As,T2Attrs),t(S,SAs,SAttrs),CB,CA),
          delete(CB,dcon(type,subt,V,t(T2,T2As,T2Attrs)),CC) )
      ; CC=CI,t(S,SAs,SAttrs)=t(T,TAs,TAttrs) ),
      ( member(dcon(type,supt,V,t(S2,S2As,S2Attrs)),CC)->
          st(t(S2,S2As,S2Attrs),t(S,SAs,SAttrs),CO,CC)
        ; (S \== bottom)->CO=CC;!,fail ) ) )
  ; !,fail.
 %% don't record supt constraints bigger or equal to  VSup
ctxupd(supt,tv(V,VSup),t(T,TAs,TAttrs),CO ,CI) :-
  ( st(VSup,t(T,TAs,TAttrs),CA,CI)->(CO=CI) )
  ;
  ((( member(dcon(type,supt,V,t(T2,T2As,T2Attrs)),CI)->
```

```
        ( join(t(T,TAs,TAttrs),t(T2,T2As,T2Attrs),t(S,SAs,SAttrs),CA,CI),
          delete(CA,dcon(type,supt,V,t(T2,T2As,T2Attrs)),CB) )
    ; CB=CI,t(S,SAs,SAttrs)=t(T,TAs,TAttrs) ),
    ( member(dcon(type,subt,V,t(S2,S2As,S2Attrs)),CB)->
          st(t(S,SAs,SAttrs),t(S2,S2As,S2Attrs),CO,CB)
      ; (S \== bottom)->CO=[dcon(type,supt,V,t(S,SAs,SAttrs)) | CB];!,fail ))
  ; !,fail) .

/*

 */
ctxrepl(_,_,[],[]).
ctxrepl(E,V,[dcon(cocontpl(N,T),OP,E2,Y) | Ys],
  [dcon(cocontpl(N2,T),OP,E2,X) | Xs]) :-
    ctxrepl(E,V,N2,N),
    ctxrepl(E,V,Y,X),
    ctxrepl(E,V,Ys,Xs).
ctxrepl(E,V,[dcon(T,OP,E2,Y) | Ys],[dcon(T,OP,E2,X) | Xs]) :-
    ctxrepl(E,V,Y,X),
    ctxrepl(E,V,Ys,Xs).
ctxrepl(E,V,[Y | Ys],[X | Xs]) :-
    ctxrepl(E,V,Y,X),ctxrepl(E,V,Ys,Xs).
ctxrepl(E,V,(Ya,Yb),(Xa,Xb)) :-
    ctxrepl(E,V,Ya,Xa),ctxrepl(E,V,Yb,Xb).
ctxrepl(E,V,V,E).  %% actually do the substitution
ctxrepl(_,_,X,X).  %% no substitution required, no action.

/*

 */
ctxeval([],X,X). %% empty context, all things are themselves.
ctxeval(CTX,tt(T),tt(TC)) :- ctxeval(CTX,T,TC).
ctxeval(CTX,t(T,TArgs,TAttrs),t(T,T2Args,T2Attrs)) :-
```

```
    ctxeval(CTX,TArgs,T2Args),ctxeval(CTX,TAttrs,T2Attrs).
ctxeval(CTX,tv(V,VSup),MST) :-
    ( member(dcon(type,supt,V,MST),CTX)
    ; member(dcon(type,subt,V,MST),CTX)
    ; MST=tv(V,VSup) ).
ctxeval(CTX,dt(cocontpl(N,T),E),dt(cocontpl(N2,T),V)) :-
    ctxeval(CTX,N,N2),ctxeval(CTX,E,V).
ctxeval(CTX,dt(T,V),dt(T,V2)) :- ctxeval(CTX,V,V2).
ctxeval(CTX,attr(AC,V),attr(AC,V2)) :- ctxeval(CTX,V,V2).
ctxeval(_,[],[]).
ctxeval(CTX, [X | Xs], [Y | Ys]) :-
    ctxeval(CTX, X, Y),ctxeval(CTX,Xs,Ys).
ctxeval(CTX, (X1,X2), (Y1,Y2)) :-
    ctxeval(CTX,X1,Y1),ctxeval(CTX,X2,Y2).
ctxeval(CTX, X, Y) :-
    atom(X),member(dcon(_,eq,X,Z),CTX),ctxeval(CTX,Z,Y).
ctxeval(_, X, X). %% X a number or not present in context

/*
```

The minimum super type of a type is itself evaluated in the context, or the meet of all super type restrictions on a type variable; if there are none in the context, then just the one in it's definition.

```
 */
minsupt(_,[],[]).
minsupt(CTX,[S|Ss],[T|Ts]) :- minsupt(CTX,S,T),minsupt(CTX,Ss,Ts).
minsupt(CTX,t(T,TAs,TAttrs),t(T2,T2As,T2Attrs)) :-
    ctxeval(CTX, t(T,TAs,TAttrs), t(T2,T2As,T2Attrs)).
minsupt(CTX,tv(V,VSup),MSupT) :-
    ctxeval(CTX, tv(V,VSup), TC),
    ( (TC=t(T,TAs,TAttrs), MSupT=t(T,TAs,TAttrs))
    ; (TC=tv(_,V2Sup),MSupT=V2Sup) ).
```

```
/*
```

Data terms have the form dt(T, N, [Vals]) where T is the type and Vals values of the supported type. Integers, booleans and reals (as floats), optionally as lists, are supported. dtcheck makes sure each element of a list is a data term of the appropriate type.

```
 */
dtscheck([],0,_).
dtscheck([dt(S,_) | Ds],N,T) :- S==T, M is (N-1), dtscheck(Ds,M,T).
dtscheck(_,_,_) :- !,fail. %% termination condition, must be last

%% tests that each element of the list is a type term,
% and right number of elements.
ttscheck([],0,_,CTX,CTX).
ttscheck([tt(t(S,Ss,SAttrs)) | Ts],N,T,CTXOut,CTXIn) :-
    st(t(S,Ss,SAttrs),T,CTXOut,CTXA), M is (N-1),
    ttscheck(Ts,M,T,CTXA,CTXIn).
ttscheck(_,_,_,_,_) :- !,fail. %% termination condition, must be last

unwrap([],[]).
unwrap([dt(_,V)|DTVs],[V|Vs]) :- unwrap(DTVs,Vs).
unwrap([tt(T) | TTVs],[T|Ts]) :- unwrap(TTVs,Ts).
unwrap([X|Xs],[X|Ys]) :- unwrap(Xs,Ys).

/*
```

```
*/
```

Chapter 7

Examples

The best way to understand how HUSC works, and the problems encountered, is to study some sample problems. Each section presents an isolated example problem, the HUSC code to solve it, and the resulting type judgements from HUSC.

In order to make it comprehensible, the output files have been altered and documented.

7.1 Long and Short Integer Arithmetic

Consider the situation where an embedded system is being developed for a new, and steadily evolving, hardware platform. The current hardware environment is an 8 bit CPU with machine instructions for both 8 and 16 bit arithmetic. The software must run on this system, but it should be as hardware independent as possible to adapt to new platforms. In this situation, the abstract model will first be developed in one module, and then used as the basis for the concrete model where appropriate restrictions are applied.

The abstract model defines the relations between the data items, in this case some simple arithmetic. It uses generic Integer and Real types for the data and operators. This system cannot be implemented as specified because there are no limits on the size of the data values, so the numbers and operations won't fit into the system, but the relationships are generally well typed.

The concrete model imports the data definitions from the abstract model. It sup-

plies concrete input definitions which introduce bounds, in the form of the Range attribute, into the abstract model. It also supplies bounded, concrete arithmetic operator definitions that describe both short and long integer arithmetic. The concrete model does not import the unbounded Integer operators, so any operation that isn't well bounded will result in a typing failure. The concrete module interface could become the program interface, or it could be imported by another module, as long as the inputs are equally or more constrained than the import declarations.

The abstract model presented here would likely be the actual program used in traditional environments, and that any bounds checking would be left to the discretion of the programmer. In HUSC, the bounded operators must include the run-time test that makes sure the operation is valid. The test is automatically eliminated if it can be shown unnecessary, so in unambiguous situations there is no run time overhead.

7.1.1 Abstract Model HUSC code

Note that in a real program, the definition of Integer and + would be supplied through other modules, making up the preludes for the language.

```
module Abs
  export (out1,out2)
  import (nin :: Integer)
{
Define Integer as Scalar;
Operator + as (i::Integer) + (j::Integer) = (k::Integer);

Let out1,out2 be Integer;
out1 = nin + 2;
out2 = nin + 128;

}
```

7.1.2 Abs Module Type Judgements

The vt predicates provide final type definitions for their variables. The type is encoded as t(*name*,*dependent expressions*,*attributes*).

There are three possible type errors that can occur:

1. Bottom means their are incompatible constraints on the identifier based on its usage, declarations and definition.
2. Ambiguous contexts result in Top types, meaning there was not enough information to judge the type.
3. No vt predicate means the type could not be determined; this usually means no satisfactory function or operator definitions could be found for some part of the expression.

The et predicates are the type judgements for individual expressions and subexpressions, but otherwise they are the same.

```
vt(out2, t(Integer, [], []))
vt(out1, t(Integer, [], []))
vt(nin, t(Integer, [], []))

et(out2@1, t(Integer, []))
et(out2@2, t(Integer, []))
et(out1@1, t(Integer, []))
et(out1@2, t(Integer, []))
et(cocoinput(nin), t(Integer, []))
et(out1@3, t(Integer, []))
et(out2@3, t(Integer, []))
```

The function / operator interfaces selected for use in an expression are recorded as joinex (join expressions). In this case, there is only one interface available for each operation. Where there are more than one, the type of the result is the join of their result types.

If there are no joinex for a function / operator expression, then no interface satisfied the type context, and the module cannot be compiled. Note that there is no requirement for the function implementation to be included in the module, these can be supplied later.

```
joinex(out2@1, +1@1, t(Integer, [], []), [])
joinex(out1@1, +1@1, t(Integer, [], []), [])
```

The data definitions are expressed as ”static” constraints. These are part of the input, and are never erased during processing. They can be used to reassemble the definitions by tracing the expression label associations. For example binop(out1@1, +, out1@2, out1@3) means $out1@1 = out1@2 + out1@3$.

```
lit(out1@3, t(Integer, [], [attr(Range, dt(interval(int), 2, 2))]))
lit(out2@3, t(Integer, [], [attr(Range, dt(interval(int), 128, 128))]))
var(out1@2, nin)
var(out2@2, nin)
binop(out1@1, +, out1@2, out1@3)
binop(out2@1, +, out2@2, out2@3)
vdecl(nin, t(Integer, [], []))
vdecl(out2, t(Integer, [], []))
vdecl(out1, t(Integer, [], []))
vdef(nin, cocoinput(nin))
vdef(out1, out1@1)
vdef(out2, out2@1)
```

The ec predicates represent shape constraints on expressions. Where there is more than one ec for an expression, they are merged by taking their meet, so there will only ever be one per expression in the output. They are a good source for information about type failures, to trace back which expression first received a Bottom constraint, the result of incompatible shape constraints.

```
ec(+1@2, t(Integer, []))
ec(+1@1, t(Function, [tt(t(Integer, [], [])), dt(int, 2), tt(t(Integer, [], [])),
ec(cocoinput(nin), t(Integer, []))
```

```
ec(out2@1, t(Integer, []))
ec(out1@1, t(Integer, []))
ec(out1@3, t(Integer, []))
ec(out1@2, t(Top, []))
ec(out2@3, t(Integer, []))
ec(out2@2, t(Top, []))
```

This is trace material left over from the processing. The attrule predicates are rules that were not satisfied, any satisfied rules would have been removed. The ac's are supported constraints, these are redundant to the type judgements for the expressions; allacs just gathers up attributes for expressions. Looking at the trace material can help sort out problems with the attributes.

```
attrule(Range, plus, [out1@1, out1@2, out1@3])
attrule(Symmetric, plus, [out1@1, out1@2, out1@3])
attrule(Range, plus, [out2@1, out2@2, out2@3])
attrule(Symmetric, plus, [out2@1, out2@2, out2@3])
ac(out1@3, attr(Range, dt(interval(int), 2, 2)))
ac(out2@3, attr(Range, dt(interval(int), 128, 128)))
allacs(out1@3, [attr(Range, dt(interval(int), 2, 2))])
allacs(out1@2, [])
allacs(out1@1, [])
allacs(out2@3, [attr(Range, dt(interval(int), 128, 128))])
allacs(out2@2, [])
allacs(out2@1, [])
```

The type system completed successfully when the Yes appears at the end. Successful completion does not imply the module is well typed, just that no errors were encountered in the inferencing system.

```
Yes
```

7.1.3 Concrete Model HUSC code

The concrete model uses the Range attribute to limit the arithmetic and data values.

```
module Conc
  export (cout1,cout2)
  import (cin :: SInt{Range(-100 .. 100)})
{
Define LInt as Integer{Range(-32767 .. 32768)};
Define SInt as LInt{Range(-127 .. 128)};
Operator + as (i::LInt) + (j::LInt) = (k::LInt);
Operator + as (i::LInt) + (j::LInt) = (k::LInt);

Let cout1 be SInt;
Let cout2 be LInt;

cout1 = out1;
cout2 = out2;
Use Abs for (Integer,out1,out2) with {nin = cin};
}
```

Note that in a real program, the definition of Integer and + would be supplied through other modules, making up the preludes for the language.

```
module Abs
  export (Integer,out1,out2)
  import (nin :: Integer)
{
Define Integer as Scalar;
-- Operator + as (i::Integer) + (j::Integer) = (k::Integer);

Let out1,out2 be Integer;
out1 = nin + 2;
out2 = nin + 128;

}
```

7.1.4 Conc Module Output

Here the output shows that the ranges are propagated through the type judgement.

The main result here is that the range limit on cout2 being a short integer (SInt) funnels down into the join selection for the out2 sum. Because cin is in range $(-100, 100)$, adding 128 to it will push it out of SInt range. Therefore out2 can only be created using the LInt addition. If $out2 = nin + 2$, as in the definition for out1, both versions of plus would be in the join, and out2 could have been implemented as a SInt.

Note too the the proof obligations for the input cin are expressed separately for output to the obligations file. Also note that the proof obligation for the Abs input variable nin has been erased, as it's definition guarantees it's an integer.

This result was manually altered to display the correct results for the purposes of providing an example. There are currently some problems in the encoding between HUSC compiled code and the type system. The results are based on unit testing with hard coded type system files.

```
vt(Abs@nin, t(Abs@Integer, [], []))
vt(cin, t(SInt, [], []))

joinex(out2@1, +1@1, t(LInt, [], []), [])
joinex(out1@1, +1@1, t(LInt, [], []), [])
joinex(out1@1, +1@1, t(SInt, [], []), [])
joincan(Abs@out2@1, +, +2@1)

xpfobl(cocoinput(cin), attr(shape, t(SInt, [])))
xpfobl(cocoinput(cin), attr(Range, dt(interval(int),(-100,100))))

et(Abs@out2@2, t(SInt, []))
et(Abs@out1@2, t(SInt, []))
et(Abs@nin@1, t(SInt, []))
et(Abs@out1@3, t(SInt, []))
et(Abs@out2@3, t(LInt, []))
et(cocoinput(cin), t(SInt, []))
et(Abs@nin, t(SInt, []))
ec(Abs@out2@1, t(LInt, []))
```

```
ec(Abs@out1@1, t(SInt, []))
ec(Abs@out1@3, t(Abs@Integer, []))
ec(Abs@out1@2, t(Abs@Integer, []))
ec(Abs@out2@3, t(Abs@Integer, []))
ec(Abs@out2@2, t(Abs@Integer, []))
ec(cocoinput(cin), t(SInt, []))
ec(cout1@1, t(SInt, []))
ec(cout2@1, t(LInt, []))
lit(Abs@out1@3, t(Integer, [], [attr(Range, dt(interval(int), 2, 2))]))
lit(Abs@out2@3, t(Integer, [], [attr(Range, dt(interval(int), 128, 128))]))
var(Abs@nin@1, cocoinput(cin))
var(Abs@out1@2, (Abs@nin))
var(Abs@out2@2, (Abs@nin))
var(cout1@1, (Abs@out1))
var(cout2@1, (Abs@out2))
binop(Abs@out1@1, +, Abs@out1@2, Abs@out1@3)
binop(Abs@out2@1, +, Abs@out2@2, Abs@out2@3)
vdecl(Abs@nin, t(Abs@Integer, [], []))
vdecl(Abs@out2, t(Abs@Integer, [], []))
vdecl(Abs@out1, t(Abs@Integer, [], []))
vdecl(cin, t(SInt, [], []))
vdecl(cout1, t(SInt, [], []))
vdecl(cout2, t(LInt, [], []))
vdef(Abs@nin, Abs@nin@1)
vdef(Abs@out1, Abs@out1@1)
vdef(Abs@out2, Abs@out2@1)
vdef(+2, +2@1)
vdef(+1, +1@1)
vdef(cin, cocoinput(cin))
vdef(cout1, cout1@1)
vdef(cout2, cout2@1)
attrule(Range, plus, [Abs@out1@1, Abs@out1@2, Abs@out1@3])
```

```
attrule(Symmetric, plus, [Abs@out1@1, Abs@out1@2, Abs@out1@3])
attrule(Range, plus, [Abs@out2@1, Abs@out2@2, Abs@out2@3])
attrule(Symmetric, plus, [Abs@out2@1, Abs@out2@2, Abs@out2@3])
ac(Abs@out1@3, attr(Range, dt(interval(int), 2, 2)))
ac(Abs@out2@3, attr(Range, dt(interval(int), 128, 128)))
ac(Abs@out2@2, attr(Range, dt(interval(int), (-100,100))))
ac(Abs@out2@1, attr(Range, dt(interval(int), (128,288))))
acs(cout1@1, attr(Range, dt(interval(int), (-127,128))))
acs(cout2@1, attr(Range, dt(interval(int), (-32767,32768))))
allacs(Abs@out1@2, [attr(Range, dt(interval(int), (-100,100)))])
allacs(Abs@out1@3, [attr(Range, dt(interval(int), 2, 2))])
allacs(Abs@out1@2, [attr(Range, dt(interval(int), (-100,100)))])
allacs(Abs@out1@1, [attr(Range, dt(interval(int), (-127,128)))])
allacs(Abs@out2@3, [attr(Range, dt(interval(int), (128, 128)))])
allacs(cout1@1, [attr(Range, dt(interval(int), (-127,128)))])
allacs(cout2@1, [attr(Range, dt(interval(int), (-32767,32768)))])

Yes
```

Chapter 8

Conclusions and Future Work

8.1 Summary

The results of the HUSC project are an early but promising start on satisfying the initial goals of the Coconut project. HUSC establishes a modularized framework for defining the fundamental relationships and types found in mathematical systems, then refining those relationships with more detailed information. The ability to define new attribute classes (even though it currently requires CHR programming) will allow the language to be customized for very specific problem sets, a key objective of the system. This is especially powerful when combined with the function specialization abilities, which allows broad classes of systems to be defined without any future knowledge of the specialized contexts they will be embedded in.

The type system is the main contribution of this project, in particular the support for function specialization and attribute classes made possible by subtyping. The decomposition of subtyping into shapes and attributes and the constraint based type inferencing make this a unique and valuable approach to static type analysis.

The HUSC type system has not been proven sound nor complete, and the implementation of the type system is rudimentary. There are a number of significant upgrades that are required, in particular in records and domain checking multiple definitions for arrays. It does, however, demonstrate that further effort is warranted in this effort, and will serve as a useful starting point for solutions in these areas.

The biggest question that remains outstanding is whether a set of attributes can

be defined that will solve practical problems in the real world. The linear algebra example given in the introduction illustrates how this might be approached, but as is always the case with the real world, the challenge is in the details. An immediate goal for proceeding forwards will be to develop a set of attribute classes, function definitions and problems that can be implemented in HUSC to test the expressiveness of the language.

One problem in particular that was not addressed was the definition and implementation of a "precision" attribute class. This is a significant problem, not least of which because the definition of precision is difficult to obtain. It is also likely that precision will require information from other attributes, such as range, which then makes proving the type system is sound and complete even harder. The problem stems from the difficulty in defining formal semantics for the floating point approximations of real numbers, a problem which should be addressed by HUSC in the future. This is a very significant factor in complex systems development, and HUSC may provide a good environment in which to explore this area.

It is worth noting that the type system can be used independently of the HUSC language and the Coconut project, although the definition given here mandates a declarative language. To make the type system more generally useful, algebraic data types would have to be incorporated as shapes or classes of shapes, an interesting problem area on its own.

The concrete syntax was intended to be a bridge between programmers familiar with Fortran and Matlab and strongly typed declarative programming languages. It is a literate language, meaning the source code is embedded in LaTeXsource files, and is intended to be further integrated with LaTeXto provide a better syntax for representing mathematics. However, neither of these were high priority goals for this project, and their suitability in this regard has not been addressed.

One general observation drawn from this work is that it is very difficult to express the broader range of constraints HUSC makes available in the idiom of a type system. The purpose of the HUSC type system is the specification of safe programs and supporting the use of function specializations in certain contexts, to allow the lower level compiler to optimize the system. It may be better to generalize the notion of the type system to be instead a constraint solver that works as a partial evaluator,

reducing expressions to their most constrained form (possibly a solution). Then new kinds of constraint solvers can be introduced without attempting to limit their domain to a particular identifier in a context. This is another possible area of future work, but falls outside the scope of HUSC.

The future work on HUSC falls into four areas:

Applications a body of application programs should be developed, in particular mathematical foundations like addition and multiplication, to test and develop the expressivity of the language and type system.

Proving Soundness and Completeness of the type system and HUSC semantic, likely establishing new restrictions on the system.

Resolving outstanding weaknesses of the current implementation, in particular the interaction between data terms and dependent type variables in the type judgment contexts, and of course pulling the type judgements back into the code hypergraphs.

Develop the Codegraph Transformer which is the next layer in the Coconut project, in order to confirm the usefulness of the model, and drive out new requirements to improve the HUSC language.

These last two points have been developed in more detail here, providing some direction for progress in those areas.

8.2 Compiling HUSC Programs

The types defined for HUSC are "pure" types, in that there is no requirement that they are representable on any real system or even representable at all. This is particularly important for modules capturing physical relationships from the real world context that may not play any role in computation, but exists solely to be checked by the type checker. It is also a handy abstraction when a system is being modelled to be implemented over a range of hardware systems.

Eventually, however, the modules must be imported into a system that provides sufficient context to build a concrete system. This requires the following conditions to be met:

1. the exported (output) definitions of the main module must be *ground*, meaning well defined and well typed.

2. each term must be of a representable type, including all intermediate results

3. each function or operator application must have an associated interface that satisfies the final type context (a successful join candidate) and there must be an implementation available for this interface.

Finally, the user of the system must guarantee that the external proof obligations are satisfied, but there is no attempt to verify this by the system.

This transformation is outside the scope of this work, and will be handled by the lower layer of the Coconut system. However, some discussion has been provided here to clarify what steps must occur and how this might work in practice, in order to assist in the understanding and verification of HUSC. These are likely not a complete set of requirements, but encapsulate the most important requirements for completion.

The output of the HUSC system is an attributed syntax hypergraph, with terms as vertices and functions / operations as edges. It is a hypergraph because the joins are represented as individual edges between nodes.

The codegraph transformer will look for patterns in the typed ASGs, and make transformations to simplify or optimize the graphs. Some categories of transformations are suggested below.

Attribute Testing

When an expression has requested an attribute be tested as a guard to a join expression, the transformer must insert the predicate defined in the attribute class definition to ensure the run time context does support the constraint. This is a requirement to support the HUSC specification.

Storage Model

HUSC types need to be transformed, at least within the theoretical model of the system, into concrete native system types For example, an Integer {Range(-10,10)} is representable as an int, a char, long int or whatever is convenient, but an Integer is not generally representable as it is unbounded (although the system may include libraries to deal with this if desired). An Array{Symmetric}[3,3] of Integer{Range(-127,128)} may be represented by a buffer with 6 or 9 1 byte elements, depending on whether keeping the symmetric attribute is considered desirable in the final result. The importance of the semantic information is clear in this case: the benefit of a symmetric model for a 3x3 array is questionable, but for a 10000 x 10000 array, it is likely to be a good choice. How these decisions will be made is clearly a difficult subject, but the fact remains that the decisions can be made now.

Ensuring that the representation of Real and Complex numbers is accurate and safe will require further analysis, particularly the implementation of a precision attribute class. This has not been done, but would be an excellent area to study.

Removing Abstract Type References

HUSC allows the introduction of abstract types which play no role in computation. For example, a matrix may be defined as an array over a basis. The basis could be left as an abstract value with only equality provided over the type, and no definition of equality provided. Matrix addition would be undefined where the bases were unequal, and simple array arithmetic when they are equal.

Since there is no definition of equality over bases, two bases can only be equal when they are identical syntactically, i.e. one is defined to be equal to the other. Since equality has the property of reflexivity, the code graph transformer could automatically replace any Boolean guard testing $\alpha == \alpha$ with True, discarding the guard and the alternate branch (undefined). If this were not the case, the module could not be compiled successfully as is appropriate.

System Input / Output and Timing Issues

HUSC is quite abstract and static in its representation of a system, providing little or no direct information about algorithms or processing times. The code block transformer will have to introduce any information about the system input and output, particularly in real time systems.

8.3 Outstanding HUSC Implementation / Design Issues

8.3.1 Context Management and Symbolic Computation

The single biggest difficult with the implementation was managing the context for dependent data terms. There were many requirements for being able to switch context between data identifiers (variables, constants) and dependent arguments in type definitions. While no one requirement was impossible, the actual implementation became quite messy as the sum of the requirements really dictates a symbolic calculator be made available within the type system.

Initially this could handle just simple scalar data types. However, eventually the type system should be able to take a data term of any HUSC type, and use the loaded HUSC operations to manipulate that property, including doing tests and arithmetic. In other words, the symbolic calculator should be a HUSC interpreter.

It is possible, perhaps even desirable, to build such a tool within the Prolog system. However, it may be preferable to rewrite the type system in Haskell, should there be a Haskell implementation of CHR that is acceptable, in which case the symbolic computation would also be done in Haskell. It is unclear which environment would be better; it is clear that this would be a difficult undertaking in either environment.

8.3.2 Building the Code Hypergraph

The type system produces type judgements for expressions, subexpressions and data identifiers, and lists of function / operator interfaces that were included in joins for operator / function application expressions. This information needs to be pulled back

into the Haskell environment and associated with the internal module data structures, then written out as the code graph.

This effort is well defined, as each expression is uniquely labelled, the relationships between the expressions are recorded explicitly, and judgement and join candidate is associated with the expression. However, merging this information represents a significant workload.

8.3.3 Attribute Class Definitions

The original syntax for attribute class definitions had propagation rules defined using the declarative syntax of HUSC. This was to be translated into the CHR system, but here were a number of difficulties with this and in the end attributes were coded directly as CHRs in the type system. At that point it was unclear what would be gained by trying to generate the syntax, so the effort was postponed to future work. The system is still fully extensible by adding the CHRs directly, so this no way diminishes the effectiveness of the type system.

As sample attribute classes were defined during the initial implementation testing, it became clear that there were kinds of attribute classes, that shared similar logical properties and would be implemented as CHRs in the same way. For example, the attributes Symmetric and Tridiagonal have identical join, meet, subattribute and propagation over addition properties. A classification system for attribute kinds would be very useful in producing a syntax to define attribute classes in a practical way that avoids the creation of boiler plate code.

Type vs. Data Terms

The HUSC language is very flexible about allowing variables to be used as dependent variables. This creates a significant difficulty when trying to encode those expressions into internal terms, as the type of the variables isn't known at the time of encoding.

The type system needs to have a translation mechanism to propagate type information into the dependent expression contexts as variables receive type judgements. This is quite tricky, but would work within the context of the CHR rules.

Again, it would be superior to have a built in HUSC interpreter that could manage

the symbolic variables and infer their types.

Join and Conditional Expressions

There are currently two different structures over expressions, the join and the conditional. The join was seen as a way of providing multiple different approaches to solve the same problem, while the conditional statement partitioned the solution space using Boolean guards. However, the resulting type of both types of statements turns out to be the same, namely a join over all of the conditions, i.e. any result that might come back from the select statement. It seems that these two structures should be merged, which would make the syntax more complex, but the selective join would be much more expressive.

The select and join statements are not currently encoded properly when being handed over to the type system. The variables in each component of the conditional expression are in effect local copies of the original, with a separate type context defined by the constraint of the Boolean guard. To encode the select statement, the variables in the guards need to removed to a local context, possibly by renaming, and then setting the result of the selection to the join of the individual types. Not renaming the variables would result in inappropriate type judgements.

For example

```
Let x be Real{Range( -1 .. 1)}
y = select {
        -x | x < 0.0
         x | otherwise
}
```

The judgement for y should clearly be Range(0 .. 1), but just joining the results of x and -x gives (-1 .. 1). Recognizing that the x in the first is an x^+ of Range (0 .. 1) isn't too difficult. What to do about the "otherwise" might be more troublesome.

Future Work on the Parser

There are a number of modifications to the parser that should be made.

1. Allow new operator symbols to be defined in modules, updating the expression parser in the Expr module.

2. make the parser token based, instead of character based, with the tokens incorporating the line numbers from the literate file.

3. Ideally a layout scheme, such as Haskell's, would be implemented as part of the deliterator.

4. extend the monad to include recording errors as they are generated (for non-terminal errors).

Parsec allows both custom tokens and state elements as part of the combinator strategy, so it should be relatively easy to incorporate these additional features.

Module System

A rudimentary module system was provided in the implementation, which uses identifier renaming to lift private definitions from imported modules into the program context while maintaining privacy syntactically. All of the definitions of the imported module are included, regardless of the actual import statement. This generates many unnecessary definitions, and will frequently result in the same definition appearing twice, with differently qualified identifiers, as two modules both import the same source module.

When a module is imported, only the definitions that are required should be imported. This requires some dependency checking algorithms, which become quite tricky because the dependent variables and attribute properties embedded in types must be considered. In addition, it is important to make sure all of the type and attribute class definitions required are imported with the data. Haskell solves this problem by always importing constructors and class definitions when you import a module, but this often creates unnecessary conflicts. It may be necessary to maintain two contexts, one for types and one for data, but there should still be scoping within the type context.

A solution for this problem will be complicated, but is a necessity for a usable Coconut system.

Miscellaneous Issues

1. add checking for cycle detection in definitions
2. The export list could be modified to include export declarations, making them mandatory for data identifiers, with some decisions being made about how to export type definitions, operators and attribute classes for which there are no declarations currently.
3. can't define new function types and then use them in declaration, e.g. Define Sinusoidal as Function ℓ; Let sin be Sinusoidal. This is a mechanical deficiency and can be resolved by chasing the base type up the type hierarchy.
4. Currently only supporting function applications for up to three parameters. This is a mechanical issue only, a little CHR coding could support arbitrary size parameter lists.

8.4 NonDeterminism, Inequality Constraints and Systems of Constraints

Traditionally programs dealing with systems of equations, often including inequality statements as well, require that these systems be transformed into strictly linear or deterministic, paths before being coded. In imperative languages, the program is transformed to include a solution suitable for the nonlinearity where the value to be solved for is moved to the LHS of the system and the remainder of the values to the RHS of the system. The transformation is assumed to be correct, and no further checking or optimization is possible because the semantic information behind the system is lost. This is nonintuitive from a mathematical point of view and opens a significant window for error.

In HUSC, the "normal" data definitions statements are also deterministic and linear. It is not possible to describe a truly nonlinear system of equations, nor the associated inequality relationships, with just these definitions. However, it should be possible to define nondeterministic, nonlinear systems of equations with expressions on both the left and right hand side of an equality or inequality symbol.

This "system of constraints" would be a data type, storing systems of equations and inequalities as a structure. A system of constraints is not evaluated by the Coconut compiler, but instead forms a complex, compound data type that is an argument to a module called a "solver". A solver accepts a system of constraints as an input definition, typically returning deterministic definitions. At the HUSC level, this becomes another form of function interface, with a different syntax, providing a definition for an identifier that is never isolated on the LHS of an equation.

Internally the system will become a compound type, with the syntax graph of the expression being the compound shape. This allows for structural type checking, but also takes advantage of the subtyping over attributes, which can be part of the interface. For example, the Crout method solves a system of linear equations that has been reduced to a tridiagonal format. The interface to that module would specify a system

```
input: sys as {$A*x = b$}; $A$ is Matrix[n,n] of $\alpha$; $b$ is Vector[n] of $\alpha$
export: x is Vector[n] of $\alpha$

-- and would then be invoked by
Solve splineCoeff as ${H*x = c}$ for $x$ using CroutReduction with ${splineCoeff}$
```

(the concrete syntax needs further development but establishes the basic idea). In this example, the module is named, but it should be possible to allow selection from a list of loaded solvers, just as the function interfaces are loaded, but this is just a conjecture at this point.

For nonlinear solvers, additional matching might be done on a cost function as an inequality and only allowing positive semidefinite matrices. This much tighter type checking, a very useful trait for type systems in this very detailed domain.

This sort of type checking is possible because of the notion of the type joins and meets. The "type" of both sides of an equality sign is the meet of the derived types of the left and right sides. In other words, the constraints on the LHS would also be applied to the RHS and vice versa. This is a simple generalization of the data definition, where constraints inferred by the usage of an identifier are applied to the expression in its definition. Since this is just one more usage of these identifiers, there is no fundamental difference to the results.

In the case of inequalities, the ability to is less clear. The meaning of the constraint imposed might be difficult to translate into attributes. There would still be type information that could be inferenced on either side in isolation, and there would have to be a definition of the inequality symbol over the types derived for the two sides, further refining the notion of their types.

Adding this capability to the HUSC language and type system should be a high priority because it promises to be both powerful and not too difficult, leveraging the strength of the constraint based type system.

Appendix A

Formal Operational Semantics

These are the formal rules defining the operational semantic that is represented by HUSC relationships, and the formal rules for the type system.

The type system is implemented using a CHR program, as described in 6, which is a direct translation of the rules into a constraint handling program.

HUSC does not provide an implementation of the operational semantics for expression evaluation. This semantics is a specification for how a back end compiler for HUSC should interpret the output code hypergraphs. An informal description of the rules is given in 3.2 The rules for *complete*definitions are intended to be implemented within HUSC, as part of the syntax checking, but this implementation is not yet complete.

A.1 Operational Semantics

In the rules below, the following conventions are used:

terms, expressions t_i are data terms, E, E_i are expressions

variables x_i, y_i, etc.

types S, T, S_i, T_i, Q, R, U

type variables α, β

attributes A, A^T, A_i^T, $A_i(\rho_i)$ where ρ are property values and T's are the type the attribute applies to

context Γ, Γ_E

sequences $x_{1..n}$ is a short form for the sequence $x_1, x_2..., x_n$

The operators $\preceq, \preceq^{Sh}, \preceq^{A}$ represent subtype, subshape and subattribute relationships respectively. Intuitively, a subtype represents a subset of the possible values that can be represented by a type, so is a more restrictive version of the type but can be used anywhere the type can be used. A subshape is the same but for the shape component of the type and a subattribute represents a more restrictive description of an attribute. The formal definitions of these operators are given in 5.2.

The symbols $\sqcup, \sqcap$ are the join and meet operators, also formally defined in 5.2. $S \sqcup T$ is the least general (most restrictive) type for which S and T are subtypes, and is used to describe the type of a join expression, where the result is one of a number of candidate expressions. Similarly $S \sqcap T$ is the most general (least restrictive) type that is a subtype of both S and T, and represents the most general unifier between two constraints on a type.

The semantic statements are given as logical axioms and predicates. An axiom is a single statement on its own. A rule consists of pre-conditions above the line, and when those conditions are satisfied, the predicate below the line is true.

The operational semantic rules describe the composition of the module context Γ, or the additional locally scoped definitions in the expression E Γ_E. Contexts are sets of relations, which include data definitions, type judgements and subtype relationships. The axioms describe what makes up a context, and the rules form an iterative definition of the contexts. In a given rules, a context may also be extended with a specific relation for the purposes of clarity ($\Gamma, x : T \vdash \ldots$). The $\vdash$symbol means "implied by context". The main module context is the abstraction of the final code hypergraph.

Expression evaluations (those labelled E—*rulename*) are given as a set of conditions that lead to a new equivalent expression for an identifier. These are to be interpreted as instructions for evaluating expressions, given their type in terms of a shape and attributes, by the next layer of the compiler system.

Definition statements (D—*rulename*) describe the rules for accepting new definitions of functions, operators, types, attribute classes and data identifiers. They use the notation $def \in \Gamma$ to indicate that a definition is present, and the notation $\Gamma \vdash def$ to indicate the definition is accepted in the context. The remaining elements of the definition rule are the conditions for accepting such a definition.

Declaration statements (C—*rulename*) are similar, but they provide a type constraint on a definition. These are represented by $x :: T$, x is declared to have type T. The double colon (::) is used to indicate a declaration in the language, which is the same as is used in the HUSC concrete syntax, the single colon (:) is a type judgement from the type system. This is a somewhat confusing convention, but the declaration :: only appears in the form $x :: T \in \Gamma$, and the meaning is very similar in any case.

There are several supplemental contexts that are separated from the main context. These are:

$\mathcal{A}$ is the attribute class context containing definitions associated with the attribute classes.

$\mathcal{F}$ is the function class context, containing the declarations of function classes and the valid function interfaces.

$\mathcal{O}$ is the operator interface context, containing valid operator interfaces for each symbol.

$\mathcal{X}$ is the external proof obligation context, representing all of the proof obligations that are created during the type evaluation of the system.

It was not necessary to separate these contexts, but the information in these is only accessible to the HUSC system, while the module context contains the information that is passed on as output. Splitting them out helps to identify what information will be dropped once HUSC has completed processing.

Finally, there is a set of rules for describing a "*complete*" definition, one which is complete with respect to the HUSC interpretation. These rules use the notation Θ to be the set of *complete*definitions, which is built recursively over the context Γ.

A.1.1 Definition, Declaration and Context Rules

$$\Gamma = \Gamma_{Output} \cup \Gamma_{In} \cup \Gamma_{Private} \qquad \text{(C—ModuleContext)}$$

$$\begin{array}{l} \Gamma = \emptyset \ \mid \ \Gamma, x : T \ \mid \ \Gamma, x = E \ \mid \ \Gamma, S = T.\{A_i(\rho_i)_{1..n}\} \\ \quad \mid \ \Gamma, \phi = (\lambda(x_i : T_i)_{1..n}.E) : T \ \mid \ \Gamma, A(\beta) \end{array} \qquad \text{(C—ContextDef)}$$

$$\Gamma_E = \Gamma \ \mid \ \Gamma_E, x : T \ \mid \ \Gamma_E, x = E_x \qquad \text{(C—Local)}$$

$$\frac{x : T \in \Gamma_{Out} \qquad T \in \mathcal{T}}{\Gamma_{Out} \vdash x : S \preceq T} \qquad \text{(C—OutputDecls)}$$

$$\frac{\Gamma_{Out} \vdash x : T \qquad x = E \in \Gamma}{\Gamma_{Out} \vdash x = E} \qquad \text{(C—OutputDefs)}$$

$$\frac{x : T \in \Gamma_{In} \qquad T \in \mathcal{T}}{\Gamma_{In} \vdash x : T} \qquad \text{(C—InputDecls)}$$

$$\frac{\Gamma_{In} \vdash x : T \qquad x = E \in \Gamma}{False} \qquad \text{(C—InputNoLocalDef)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : S \preceq T} \qquad \text{(C—Decls)}$$

$$\frac{x : T \in \Gamma_E}{\Gamma_E \vdash x : S \preceq T} \qquad \text{(C—LocalDecl)}$$

$$\frac{x = E \in \Gamma \qquad x = E' \notin \Gamma \setminus \{x = E\}}{\Gamma \vdash x = E} \qquad \text{(D—Total)}$$

$$\frac{x.l_{1..n} = E_{1..n} \in \Gamma \qquad l_i = l_j \iff i = j}{\Gamma \vdash x = (l_i = E_i)_{1..n}} \qquad \text{(D—RecPartial)}$$

$$\frac{x[i_{1..n}] = \lambda i_{1..n}.E \in \Gamma \qquad \Gamma_E \vdash i_j \in dom_j \subset \mathcal{Z}, 1 \leq j \leq n \quad \{\forall(k_{1..n}) \in dom_E\}.x[k_{1..n}] = E' \notin \Gamma \setminus \{x[k_{1..n}] = \lambda i_{1..n}.E\} \text{ where } dom_E = dom_1 \times ... \times dom_n}{\Gamma \vdash \{\forall(k_1, ..., k_n) \in dom_E\}.x[k_{1..n}] = \lambda k_{1..n}.E}$$

(D—DimPartial)

$$\frac{context \vdash \{\forall(k_1, ..., k_n) \in dom_j\}.x[i_1, ..., i_n] = \lambda i_1, ..., i_n.E_j,\ 1 \leq j \leq m \quad context \vdash x : T \qquad \mathrm{Sh}(T) = \mathrm{DimShape}(dom, \alpha)}{dom = \bigcup dom_{1..m} \qquad \exists l_{1..n}, u_{1..n} \in \mathcal{Z} \ni dom = \times[l_i..u_i]_{1..n}}$$

(D—DimTotal)

$$\frac{f(x_i : S_i)_{1..n} = E : Q \in \Gamma \qquad \mathcal{F} \vdash \mathrm{Sh}(f) = Function(T_{1..n}, R) \quad (S_i \preceq T_i)_{1..n} \qquad Q \preceq R}{\mathcal{F} \vdash f_{S_{1..n},Q} = E : Q}$$

(D—FuncInterface)

$$\frac{\mathcal{F} \vdash f_{S_{1..n},Q} = E : Q \qquad \Theta(E)}{\Gamma \vdash f_{S_{1..n},Q} = E : Q}$$

(D—FuncImplement)

$$\frac{\phi(x_i : T_i)_{1..n} = E : R \in \Gamma \qquad \mathrm{Symbol}(\phi)}{\mathcal{F} \vdash \phi_{T_{1..n},R} = E : R}$$

(D—OpInterface)

$$\frac{\mathcal{F} \vdash \phi_{T_{1..n},R} = E : R \qquad \Theta(E)}{\Gamma \vdash \phi_{T_{1..n},R} = E : R}$$

(D—OpImplement)

$$\frac{\mathrm{use(ModA, with}(x_i = E_i)_{1..n}, \mathrm{import}(y_{1..m})) \in \Gamma \qquad y_j = F'_j \notin \Gamma \qquad \Gamma_{ModA,Out} \vdash y_j : Q_j, y_j = F_j \quad \Gamma \vdash E_{1..n} : S_{1..n} \qquad \Gamma_{ModA,In} = \{x_i : T_i\}_{1..n} \qquad S_i \preceq T_i}{\Gamma \vdash y_j = F_j \qquad \Gamma \vdash y_j : R_j \qquad R_j \preceq Q_j}$$

(C—ImportDefs)

A.1.2 Simple Expression Evaluation

$$\frac{\Gamma \vdash x = E_1 \qquad \Gamma \vdash y = \lambda x.E_2}{\Gamma \vdash y = [x \mapsto E_1]E_2}$$

(E—VarEval)

$$\frac{\Gamma \vdash x = (l_i = E_i)_{1..n} \qquad \Gamma \vdash y = \lambda x.E}{\Gamma \vdash y = [x \mapsto (l_i = E_i)_{1..n}]E} \quad \text{(E—Record)}$$

$$\frac{\Gamma \vdash x = (l_i = E_i)_{1..n} \qquad \Gamma \vdash y = \lambda(x.l_j).E}{\Gamma \vdash y = [x.l_j \mapsto E_j]E} \quad \text{(E—RecField)}$$

$$\frac{\Gamma \vdash (x_{1..n}) = (E_{1..n}) \qquad \Gamma \vdash y = \lambda x.E}{\Gamma \vdash y = [x \mapsto (E_{1..n})]E} \quad \text{(E—Tuple)}$$

$$\frac{\Gamma \vdash (x_{1..n}) = (E_{1..n}) \qquad \Gamma \vdash y = \lambda(x_j).E}{\Gamma \vdash y = [x_j \mapsto E_j]E} \quad \text{(E—TupleField)}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : T \qquad \mathrm{Sh}(T) = \mathrm{DimShape}(dom, \alpha) \qquad dom = \bigcup dom_j \\ y = \lambda x[ind_{1..n}].E \qquad (ind_{1..n}) \in dom_j \qquad \Gamma \vdash (\forall k_{1..n} \in dom_j).x[k_{1..n}] = \lambda k_{1..n}.E_j\end{array}}{y = [x[ind_{1..n}] \mapsto E_j(ind_{1..n})].E} \quad \text{(E—ArrayIndex)}$$

$$\frac{\begin{array}{c}y = f(t_{1..n}) \in \Gamma_E \qquad \Gamma \vdash \mathrm{Sh}(f) = Function(T_{1..n}, R) \qquad \Gamma_E \vdash (t_i : S_i)_{1..n} \qquad (S_i \preceq T_i)_{1..n} \\ \Gamma, y : U \vdash U \preceq Q_{1..m} \qquad \bot_T \prec Q = \sqcap Q_{1..m}\end{array}}{\Gamma \vdash y = \bigwedge \{\forall S'_{1..n}, R' \ni f_{S'_{1..n},R'} \in \mathcal{F} \wedge (S_i \preceq S'_i)_{1..n} \wedge \bot_T \prec (R' \sqcap Q)\}.f_{S'_{1..n},R'}(t_{1..n})} \quad \text{(E—FuncJoin)}$$

$$\frac{\mathcal{F}_f \vdash f_{S_{1..n},R} = \lambda(p_i : S_i)_{1..n}.E_{S_{1..n},R} \qquad \Gamma_E \vdash x = f_{S,R}(t_{1..n}) \qquad \Gamma_E \vdash (t_i : S'_i)_{1..n} \qquad (S'_i \preceq S_i)_{1..n}}{\Gamma_E \vdash x = [(t_i \mapsto p_i)_{1..n}]E_{S_{1..n},R}} \quad \text{(E—FuncApplication)}$$

$$\frac{\begin{array}{c}y = \phi(t_{1..n}) \in \Gamma_E \qquad \Gamma_E \vdash (t_i : S_i)_{1..n} \\ \Gamma_E, y : R \vdash R \preceq Q_{1..m} \qquad \bot_T \prec Q = \sqcap Q_{1..m}\end{array}}{\Gamma \vdash y = \bigwedge \{\forall \phi_{(T_{1..n}),R'} \in \mathcal{O}\phi \ni (S_i \preceq T_i)_{1..n} \wedge \bot_T \prec (R' meet Q)\}.\phi_{(T_{1..n}),R'}(t_{1..n})} \quad \text{(E—OpJoin)}$$

$$\frac{\mathcal{O}_\phi \vdash \phi_{(T_{1..n}),R} = \lambda(p_i : T_i)_{1..n}.E_{T_{1..n},R} \qquad \Gamma_E \vdash x = \phi_{T,R}(t_{1..n}) \qquad \Gamma_E \vdash (t_i : S_i)_{1..n} \qquad (S_i \preceq T_i)_{1..n}}{\Gamma \vdash x = [(t_i \mapsto p_i)_{1..n}].E_{T_{1..n},R}} \quad \text{(E—OpImplement)}$$

A.1.3 Completeness Rules

The expression $FV(c)$ refers to the free variables of an expression c, and the symbols Θ as the set of complete definitions, with $\Theta(x)$ indicating x is complete.

$$\Theta \subseteq \Gamma \qquad \Theta(x) \implies x \in \Theta \tag{Θ— Definition}$$

$$\frac{\Gamma_c \vdash c : T \qquad \Gamma \vdash FV(c) = \emptyset}{\Gamma \vdash \Theta(c)} \tag{Θ— Constant}$$

$$\frac{\Gamma_{In} \vdash v : T}{\Gamma \vdash \Theta(v)} \tag{Θ— Input}$$

$$\frac{\Gamma \vdash x = v \qquad \Gamma \vdash FV(v) = \{v\} \qquad \Gamma \vdash \Theta(v)}{\Gamma \vdash \Theta(x)} \tag{Θ— Var}$$

$$\frac{\Gamma \vdash x = (l_i = E_i)_{1..n} : T \qquad \Gamma \vdash \Theta(E_i)}{\Gamma \vdash \Theta(x)} \tag{Θ— RecordDef}$$

$$\frac{\Gamma_E \vdash x = (l_i = E_i)_{1..n} : T \qquad \Gamma \vdash \Theta(x)}{\Gamma \vdash \Theta(x.l_{1..n})} \tag{Θ— RecordField}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\forall(k_{1..n}) \in dom_{E_j}\}.x[k_{1..n}] = \lambda k_{1..n}.E_j 1 \leq j \leq m \qquad \Gamma \vdash x : T \\ \mathrm{Sh}(T) = \mathrm{DimShape}(dom, \alpha), dom = \bigcup E_j \qquad \Gamma_{E_j} \vdash \Theta(E_j)\ 1 \leq j \leq m\end{array}}{\Gamma \vdash \Theta(x)} \tag{Θ— DimDef}$$

$$\frac{\Gamma \vdash x : T \qquad \mathrm{Sh}(T) = \mathrm{DimShape}(dom, \alpha), dom \subset \mathcal{Z}^n \qquad \Gamma \vdash i \in dom}{\Gamma \vdash \Theta(x[i])} \tag{Θ— DimElem}$$

$$\frac{\mathcal{F}_f \vdash f_{S,R}(t_{1..n}) = \lambda t_{1..n} : E \qquad \Gamma_E, \Theta(t_{1..n}) \vdash \Theta(E)}{\Gamma \vdash \Theta(f_{S,R})} \tag{Θ— FuncDef}$$

$$\frac{\Gamma_E \vdash x = f_{S,R}(t_{1..n}) \qquad \Gamma \vdash \Theta(f_{S,R}) \qquad \Gamma_E \vdash \Theta(t_{1..n})}{\Gamma \vdash \Theta(x)} \tag{Θ— FuncAppl}$$

$$\frac{\mathcal{O}_\phi \vdash \phi_{T_{1..n},R}(t_i : T_i)_{1..n} = \lambda t_{1..n} : E \qquad \Gamma_E, \Theta(t_1), ..., \Theta(t_n) \vdash \Theta(E)}{\Gamma \vdash \Theta(\phi_{T_{1..n},R})} \quad (\Theta\text{— OpDef})$$

$$\frac{\Gamma \vdash \Theta(\phi_{T_{1..n},R}) \qquad \Gamma_E \vdash E = \phi_{T_{1..n},R}(t_{1..n}) : R \qquad \Gamma_E \vdash \Theta(t_{1..n})}{\Gamma \vdash \Theta(E)} \quad (\Theta\text{— OpEval})$$

$$\frac{\Gamma \vdash E = Conditional.\{(G_i, E_i)_{1..n}\} \qquad \Gamma \vdash \Theta(G_{1..n}), \Theta(E_{1..n})}{\Gamma \vdash \Theta(E)} \quad (\Theta\text{— CondExpr})$$

$$\frac{\Gamma \vdash E = \bigwedge .\{E_{1..n}\} \qquad \Gamma \vdash \Theta(E_{1..n})}{\Gamma \vdash \Theta(E)} \quad (\Theta\text{— JoinExpr})$$

Note that the commutivity of the join and meet operations eliminates the need for $T \preceq S$ rules. These mirror image rules were necessary in the implementation because of the difficulties CHR has with commutative substitutions.

A.2 Type System Semantics

These rules define the type system, including shapes and attributes, subtypes, joins and meets and attribute class definitions. The approach and notation was taken from [10]. It is assumed that the reader is familiar with formal type theory descriptions, and with the general meaning of types, subtypes, dependent types. If not, [10] provides an excellent introduction to the theory of type systems in programming languages.

The following conventions are used:

- $\mathcal{T}$ is the set of valid types
- S, T, U are types;
- *Shapes* is the set of valid shape expressions
- $\text{Sh}(T)$ is the shape of type T;
- $\mathcal{A}$ is the context of defined attribute classes, part of the overall module context

- $A_i^T(\rho)$ is an attribute of class A that is applicable to type T with a property value ρ of an appropriate type.
- A_i^T is an attribute of class A that is applicable to type T, where the property value has not been represented for convenience, but does exist.
- as before, $A_{1..n}$ represents the sequence $A_1, ..., A_n$, and this notation is sometimes used "flexibly" to represent the obvious expansion of a sequence with more than one related elements (e.g $x_{1..n} :: T_{1..n}$ is $x_1 : T_1, etc.$)
- the subtype relation is not strict, so "a subtype" is equivalent to "a type and its subtypes"
- The operators $\sqcup T_i$ and $\sqcap S_i$ consolidate multiple joins and meets, respectively.

A.2.1 Fundamental Types, Top and Bottom

$$\top_T \in \mathcal{T} \qquad \bot_T \in \mathcal{T} \tag{T–TopBottom}$$

$$T \in \mathcal{T} \iff T \preceq \top_T \tag{T—Top}$$

$$T \in \mathcal{T} \iff \bot_T \preceq T \tag{T—Bottom}$$

$$\frac{T \in \mathcal{T} \qquad T \notin \{\top_T, \bot_T\}}{\begin{array}{c}\mathrm{Sh}(T) \in \mathit{Shapes} \qquad \mathrm{Attr}(T) = \varnothing \mid \{A_i^T(\rho_i)\}_{1..n} \\ \rho_{1..n} \preceq A_{1..n}^T.property \qquad T \preceq A_{1..n}^T.target\end{array}} \tag{T—GenericComp}$$

A.2.2 Shapes

$$\mathit{Shapes} = Scalars \cup Records \cup Arrays \cup Iteratives \cup Functions \tag{T—Shapes}$$

$$Tuples = \{\forall T_i \in \mathcal{T}, \forall\ n \geq 2).(T_{1..n})\} \tag{T—Tuples}$$

$$Records = \{(\forall T_i \in \mathcal{T}).(l_{1..n} :: T_{1..n}),\ l_i = lj \iff i = j\} \tag{T—Records}$$

$$Functions = \{(\forall T_i \in \mathcal{T}, \forall R \in \mathcal{T}, \forall n \geq 1).(f, T_{1..n}, R)\} \tag{T—Functions}$$

$$DimType = \{(\forall dom \in \mathcal{D}, \alpha \in \mathcal{T}).(dom, \alpha)\} \tag{T—DimType}$$

$$\frac{dom \in \mathcal{D}}{\exists l_{1..n}, u_{1..n} \in \mathcal{Z}.dom = [l_1..u_1] \times ... \times [l_n..u_n]) \qquad [l_i..u_i] \triangleq \{l_i, l_i + 1, \ell u_i\}}$$
(T—DimDomain)

$$\frac{S \in Shapes}{S = S} \qquad \text{(T—EqShapeReflex)}$$

$$\frac{\forall S1, S2 \in Shapes \qquad S1 = S2}{S2 = S1} \qquad \text{(T—EqShapeSym)}$$

$$\frac{\forall S1, S2, S3 \in Shapes \qquad (S1 = S2 \wedge S2 = S3)}{S1 = S3} \text{(T—EqShapeTrans)}$$

$$\frac{\forall S1, S2 \in Shapes.S1 = S2}{S1, S2 \in \text{same shape category}} \qquad \text{(T—EqShapeComp)}$$

$$\frac{\forall S1, S2 \in Scalar}{S1 = S2} \qquad \text{(T—EqScalar)}$$

$$\frac{\forall S1, S2 \in Tuple.S1 = S2 \qquad S1 = (T_{1..n}) \qquad S2 = (U_{1..n})}{T_{1..n} = U_{1..n}} \text{(T—EqTuple)}$$

$$\frac{\forall S1, S2 \in Record.S1 = S2 \qquad S1 = (l1_{1..n} :: T_{1..n}) \qquad S2 = (l2_{1..n} :: U_{1..n})}{l1_{1..n} = l2_{1..n} \qquad T_{1..n} = U_{1..n}}$$
(T—EqRecord)

$$\frac{\forall S1, S2 \in Function.S1 = S2 \qquad S1 = (F, (T_{1..n}), R) \qquad S2 = (F, U_{1..n}, Q)}{T_{1..n} = U_{1..n}, R = Q}$$
(T—EqFunc)

$$\frac{\forall S1, S2 \in DimShape.S1 = S2 \qquad S1 = (dom1, T) \qquad S2 = (dom2, U)}{dom1 = dom2, T = U}$$
(T—EqDim)

A.2.3 Attributes

$$\text{Sh}(T).\{A^T_{1..n}\} = \text{Sh}(T).\{A^T_{i_{1..n}}\}\text{for all permutations}\{i_{1..n}\} \quad (\text{T—AttrPermutation})$$

$$\mathcal{A} ::= \emptyset \ \mid \ \mathcal{A},\ A(\rho, \beta, \preceq^A, \sqcap^A, \sqcup^A, \models^A, rules(A)) \qquad (\mathcal{A}\text{—New})$$

$$\frac{\begin{array}{c} A(\rho, \beta, \models^A, \preceq^A, \sqcap^A, \sqcup^A, \text{rules}(A)) \in \Gamma \qquad A \notin \mathcal{A} \qquad \Gamma \implies \rho, \beta \in \mathcal{T} \\ \models^A :: \beta \to \rho \to \ boolean \in \Gamma_A \qquad \preceq^A :: \rho \to \rho \to \mathcal{B} \in \Gamma_A \\ \sqcap^A :: \rho \to \rho \to \rho \in \Gamma_A \qquad \sqcup^A :: \rho \to \rho \to \rho \in \Gamma_A \\ A :: \beta \to \rho \in \Gamma_A \qquad \text{rules}(A) \in \Gamma_A \end{array}}{\mathcal{A} \vdash A(\rho, \beta, \models^A, \preceq^A, \sqcap^A, \sqcup^A, \text{rules}(A))} \quad (\mathcal{A}\text{—Def})$$

A.2.4 Defining New Types

$$\frac{\begin{array}{c} \Gamma \vdash T \in \mathcal{T} \qquad Attributes(T) = \{A^T_{1..k}(\rho_{1..k})\} \cup \{A^T_{k+1..n}(\rho_{k+1..n})\} \\ U(\rho_{1..p}) = \Pi(\rho_{1..p})(T.((subtype?), \{A^T_{1..k}(\mu_{1..k})\} \cup \{A^U_{n+1..m}(\mu_{n+1..m})\})) \in \Gamma \qquad U \notin \Gamma \\ \rho_{1..p} \end{array}}{\begin{array}{c} \Gamma \vdash U(\rho_{1..p}) = \Pi(\rho_{1..p})(\text{Sh}(T).\{A^T_i(\rho_i) \sqcap \{A^T_i(\mu_i)\}_{1..k} \cup \{A^T_{k+1..n}(\rho_{k+1..n})\} \cup \{A^U_{n+1..m}(\mu_{n+1..m})\}) \\ \text{BaseType}\, U = T \qquad \Gamma, (subtype?) \vdash U \preceq T \end{array}}$$

(DefineType)

A.2.5 Subtypes, SubShape and SubAttribute Relations

$$T \preceq T \qquad (\text{ST—Refl})$$

$$\frac{\Gamma \vdash S \preceq T \qquad \Gamma \vdash T \preceq U}{\Gamma \vdash S \preceq U} \qquad (\text{ST—Trans})$$

$$\frac{S \preceq T \qquad T \preceq S}{S = T} \qquad (\text{ST—AntiSym})$$

$$\frac{S \prec T}{T \not\prec S} \qquad (\text{SST—ASym})$$

$$\frac{\Gamma \vdash S \prec T \qquad \Gamma \vdash T \prec U}{\Gamma \vdash S \prec U} \qquad (\text{SST—Trans})$$

$$\frac{\Gamma \vdash S \preceq T \qquad T = \mathrm{Sh}(T).\{A^T_{1..n}(\rho_{1..n})\}}{\begin{array}{c}\Gamma \vdash S = \mathrm{Sh}(S).\{A^T_{1..n}(\mu_{1..n})\} \cup \{A^S_{n+1..m}(\mu_{n+1..m})\} \\ \mathrm{Sh}(S) \preceq^{Sh} \mathrm{Sh}(T) \qquad A^T_{1..n}(\mu_{1..n}) \preceq_{A_T} A^T_{1..n}(\rho_{1..n})\end{array}} \text{(ST—Generic)}$$

$$\frac{T \in \mathit{Scalar} \qquad S \preceq T}{S \in \mathit{Scalar}} \text{(SS—Scalar)}$$

$$\frac{T = \mathit{Record}(l_1 : T_1, ..., l_n : T_n) \qquad l_i = l_j \iff i = j \qquad \Gamma \vdash S \preceq T}{S = \mathit{Record}(l_{1..n} : S_{1..n}, ..., l_m : S_m), m \geq n \qquad (S_i \preceq T_i)_{1..n}} \text{(SS—Record)}$$

$$\frac{T = \mathit{Tuple}(T_1, ..., T_n) \qquad \Gamma \vdash S \preceq T}{S = \mathit{Tuple}(S_{1..n}, ..., S_m), m \geq n \qquad (S_i \preceq T_i)_{1..n}} \text{(SS—Tuple)}$$

$$\frac{T = \mathit{Dim}(\mathit{dom}, \alpha) \qquad S \preceq^{Sh} T}{S = \mathit{Dim}(\mathit{dom}, \beta) \mathit{where} \beta \preceq \alpha} \text{(SS—Dim)}$$

$$\frac{F = \mathit{Function}(T_{1..n}, R) \qquad G \preceq^{Sh} F}{G = \mathit{Function}(S_{1..n}, Q) \mathit{where} S_{1..n} \preceq T_{1..n} \wedge Q \preceq R} \text{(SS—Function)}$$

A.2.6 Joins and Meets

$$S \sqcup T = T \sqcup S \qquad \text{(ST—JoinCom)}$$

$$S \sqcap T = T \sqcap S \qquad \text{(ST—MeetCom)}$$

$$S \sqcup (T \sqcap S) = S \qquad \text{(ST—JoinAbsorb)}$$

$$S \sqcap (T \sqcup S) = T \qquad \text{(ST—MeetAbsort)}$$

$$T \sqcup T = T \qquad \text{(ST—JoinIdem)}$$

$$T \sqcap T = T \qquad \text{(ST—MeetIdem)}$$

$$T \sqcup \top_T = \top_T \qquad \text{(ST—JoinTop)}$$

$$T \sqcap \top_T = T \qquad \text{(ST—MeetTop)}$$

$$T \sqcup \bot_T = T \qquad \text{(ST—JoinBot)}$$

$$T \sqcap \bot_T = \bot_T \qquad \text{(ST—MeetBot)}$$

$$\frac{U = S \sqcup T}{S \preceq U, T \preceq U \qquad \forall U'.(S \preceq U' \land T \preceq U') \implies U \preceq U'} \text{(ST—JoinLGG)}$$

$$\frac{U = S \sqcap T}{U \preceq S, U \preceq T \qquad \forall U'.(U' \preceq T \land U' \preceq S) \implies U' \preceq U} \text{(ST—MeetMGU)}$$

$$\frac{U = S \sqcup T}{\text{Sh}(U) = \text{Sh}(S) \sqcup \text{Sh}(T) \qquad \text{Attr}(U) = \text{Attr}(S) \sqcup \text{Attr}(T)} \text{(ST—Join)}$$

$$\frac{R = S \sqcap T}{\text{Sh}(U) = \text{Sh}(S) \sqcap \text{Sh}(T) \qquad \text{Attr}(U) = \text{Attr}(S) \sqcap \text{Attr}(T)} \text{(ST—Meet)}$$

$$\sqcup T_i = T_1 \sqcup T_2 \sqcup ... \sqcup T_n \qquad \text{(ST—BigJoin)}$$

$$\sqcap T_i = T_1 \sqcap T_2 \sqcap ... \sqcap T_n \qquad \text{(ST—BigMeet)}$$

$$\forall S, T \in \mathcal{T}. \exists U. S \preceq U, T \preceq U \qquad \text{(ST—JoinExists)}$$

$$\frac{\Gamma \vdash U = S \sqcup T \qquad S = \top_T}{\Gamma \vdash U = \top_T} \qquad \text{(ST—JoinNamedTop)}$$

$$\frac{\Gamma \vdash U = S \sqcup T \qquad S \preceq T}{\Gamma \vdash U = T} \qquad \text{(ST—JoinNamedST)}$$

$$\frac{\Gamma \vdash U = S \sqcup T \qquad S \npreceq T, T \npreceq S \qquad S \neq \top_T}{\Gamma \vdash U = (\text{BaseType}(S)) \sqcup T} \text{(ST—JoinNamedBT)}$$

$$\forall S, T \in Types. \exists U. U \preceq S, U \preceq T \qquad \text{(ST—MeetExists)}$$

$$\frac{\Gamma \vdash U = S \sqcap T \qquad S = \bot_T}{\Gamma \vdash U = \bot_T} \qquad \text{(ST—meetNamedTop)}$$

$$\frac{\Gamma \vdash U = S \sqcap T \qquad S \preceq T}{\Gamma \vdash U = S} \quad (\text{ST---}\textsc{meetNamedST})$$

$$\frac{\Gamma \vdash U = S \sqcap T \qquad S \not\preceq T, T \not\preceq S}{\Gamma \vdash U = \bot_T} \quad (\text{ST---}\textsc{meetNamedBT})$$

Joins / Meets over Shapes

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Scalar} = \mathrm{Sh}(T)}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Scalar}} \quad (\text{ST---}\textsc{JoinScalar})$$

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Scalar} = \mathrm{Sh}(T)}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Scalar}} \quad (\text{ST---}\textsc{MeetScalar})$$

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Tuple}(S_{1..n}), \mathrm{Sh}(T) = \mathit{Tuple}(T_{1..n})}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Tuple}(S_{1..n} \sqcup T_{1..n})}$$
(ST—JoinTuple)

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Tuple}(S_{1..n}), \mathrm{Sh}(T) = \mathit{Tuple}(T_{1..n})}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Tuple}(S_{1..n} \sqcap T_{1..n})}$$
(ST—MeetTuple)

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Record}(l_{1..n} :: S_{1..n}), \mathrm{Sh}(T) = \mathit{Record}(l_{1..n} :: T_{1..n})}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Record}(l_i :: (S_i \sqcup T_i))_{1..n}{}^{1}}$$
(ST—JoinRecord)

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Record}(l_{1..n} :: S_{1..n}), \mathrm{Sh}(T) = \mathit{Record}(l_{1..n} :: T_{1..n})}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Record}(l_i :: (S_i \sqcap T_i))_{1..n}}$$
(ST—MeetRecord)

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Function}(F, S_{1..n}, Q), \mathrm{Sh}(T) = \mathit{Function}(F, T_{1..n}, R)}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Function}(F, (S_i \sqcup T_i)_{1..n}, Q \sqcap R)}$$
(ST—JoinFunc)

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Function}(F, S_{1..n}, Q), \mathrm{Sh}(T) = \mathit{Function}(F, T_{1..n}, R)}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Function}(F, (S_i \sqcap T_i)_{1..n}, Q \sqcap R)}$$
(ST—MeetFunc)

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Dim}(\mathit{dom}_S, \alpha_S), \mathrm{Sh}(T) = \mathit{Dim}(\mathit{dom}_T, \alpha_T) \qquad \mathit{dom}_S = \mathit{dom}_T}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Dim}(\mathit{dom}_S, \alpha_S \sqcup \alpha_T)}$$
(ST—JoinDim)

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) = \mathit{Dim}(\mathit{dom}_S, \alpha_S), \mathrm{Sh}(T) = \mathit{Dim}(\mathit{dom}_T, \alpha_T) \qquad \mathit{dom}_S = \mathit{dom}_T}{\Gamma \vdash \mathrm{Sh}(U) = \mathit{Dim}(\mathit{dom}_S, \alpha_S \sqcap \alpha_T)}$$
(ST—MeetDim)

$$\frac{\Gamma \vdash U = S \sqcup T \qquad \Gamma \vdash \mathrm{Sh}(S) \sqcup \mathrm{Sh}(T) = \top_T}{\Gamma \vdash U = \top_T} \text{ (ST—JoinShapeTop)}$$

$$\frac{\Gamma \vdash U = S \sqcap T \qquad \Gamma \vdash \mathrm{Sh}(S) \neq \mathrm{Sh}(T)}{\Gamma \vdash U = \bot_T} \text{ (ST—MeetShapeBot)}$$

Joins / Meets over Attributes

$$\frac{\begin{array}{c}\mathrm{Attr}(U) = \mathrm{Attr}(S) \sqcup \mathrm{Attr}(T) \\ \mathrm{Attr}(S) = \{A_{1..n}(\rho_{S,1..n})\} \cup \{A^S_{n+1..ms}\} \qquad \mathrm{Attr}(T) = \{A_{1..n}(\rho_{T,1..n})\} \cup \{A^T_{n+1..mt}\}\end{array}}{\mathrm{Attr}(U) = \{A_i(\rho_{S,i}) \sqcup A_i(\rho T, i)\}_{1..n}}$$
(ST—JoinAttributes)

$$\frac{\begin{array}{c}\mathrm{Attr}(U) = \mathrm{Attr}(S) \sqcup \mathrm{Attr}(T) \\ \mathrm{Attr}(S) = \{A_{1..n}(\rho_{S,1..n})\} \cup \{A^S_{n+1..ms}\} \qquad \mathrm{Attr}(T) = \{A_{1..n}(\rho_{T,1..n})\} \cup \{A^T_{n+1..mt}\}\end{array}}{\mathrm{Attr}(U) = \{A_i(\rho_{S,i}) \sqcap A_i(\rho T, i)\}_{1..n} \cup \{A^S_{n+1..ms}\} \cup \{A^T_{n+1..mt}\}}$$
(ST—MeetAttributes)

A.2.7 Type Judgements

Declarations (except Functions)

$$\frac{\begin{array}{c}x :: T \in \Gamma \qquad T \in \Gamma \qquad T = \mathrm{Sh}(T).\{A^T_{1..n}(\rho_{1..n})\} \\ \mathcal{A} \vdash T \preceq A^T_{1..n}.\mathit{target} \qquad \rho_{1..n} \preceq A^T_{1..n}.\mathit{property}\end{array}}{\Gamma \vdash \exists S \in \mathcal{T}.S \preceq T \wedge x :: S} \text{ (T—VarDecl)}$$

$$\frac{x :: T \in \Gamma \qquad \mathrm{Sh}(T) = \mathit{Record}(l_{1..n} : T_{1..n}) \qquad \Gamma \vdash x.l_{1..n} = E_{1..n}}{\Gamma \vdash \exists S_i \in \mathcal{T}.S_i \preceq T_i.E_i :: S_i \mathrm{for} 1 \leq i \leq n}$$
(T—RecDeclFields)

$$\frac{x :: T \in \Gamma \qquad \text{Sh}(T) = Tuple(T_{1..n}) \qquad \Gamma \vdash (x_{1..n}) = (E_{1..n})}{\Gamma \vdash \exists S_i \in \mathcal{T}.S_i \preceq T_i.E_i :: S_i \text{for} 1 \leq i \leq n} \text{(T—TupDeclElems)}$$

$$\frac{x :: T \in \Gamma \qquad \text{Sh}(T) = Dim(dom, \alpha_T) \qquad \Gamma \vdash x = E}{\Gamma \vdash \exists S \in \mathcal{T}.S \preceq T.E :: S} \text{(T—DimDeclTotal)}$$

$$\frac{\begin{array}{c} x :: T \in \Gamma \qquad \text{Sh}(T) = Dim(dom, \alpha_T) \qquad dom = \times_{(doml_{1..n}..domu_{1..n})} \\ \Gamma \vdash \forall (ind_{1..m}) \in (l_{1..m}..u_{1..m}), m \leq n.x(k_{1..n}) = \lambda(k_{1..n}).E_A \qquad \Gamma \vdash k_j = \lambda(ind_{1..m}).K_j \end{array}}{\begin{array}{c} \Gamma \vdash \exists \alpha_S \in \mathcal{T}.\alpha_S \preceq \alpha_T.E_A :: \alpha_S \\ \Gamma \vdash ind_{1..m} :: Integer\{Range(l_{1..m}..u_{1..m})\} \qquad \Gamma \vdash k_{1..n} :: Integer\{Range(doml_{1..n}..domu_{1..n})\} \end{array}}$$
(T—DimDeclPartial)

A.2.8 Definitions (except Functions)

$$\frac{\Gamma \vdash x = E \qquad \Gamma_E \vdash E : S}{\Gamma \vdash \exists T \in Types.S \preceq T.x :: T} \qquad \text{(T—DefComplete)}$$

$$\frac{\Gamma \vdash x = E \qquad \Gamma \vdash E = (E_i)_{1..n} \qquad \Gamma_E \vdash (E_i :: S_i)_{1..n}}{\Gamma \vdash \exists T_{1..n} \in Types.(S_i \preceq T_i).x :: Tuple(T_{1..n})} \text{(T—DefTupTotal)}$$

$$\frac{\Gamma \vdash x = E \qquad \Gamma \vdash E = (l_i = E_i)_{1..n} \qquad \Gamma_E \vdash (E_i :: S_i)_{1..n}}{\Gamma \vdash \exists T_{1..n} \in Types.(S_i \preceq T_i).x :: Record(l_{1..n} : T_{1..n})} \text{(T—DefRecTotal)}$$

$$\frac{\Gamma \vdash x.l_1 = E_1 \qquad \Gamma_{E_1} \vdash E_1 : S_1}{\Gamma \vdash \exists T_1, T \in Types.S_1 \preceq T_1.Shapeof(T) = Record(l_1 :: T_1, l_{2..n} :: T_{2..n})}$$
(T—DefRecPartialDecl)

$$\frac{\begin{array}{c} \Gamma \vdash \{\forall (k_1, ..., k_n) \in dom_{E_j}\}.x[i_1, ..., i_n] = \lambda i_1, ..., i_n.E_j \\ \Gamma_{E_j} \vdash E_j : S_j \qquad \bot_T \prec S = \sqcup S_j \end{array}}{\Gamma \vdash \exists \alpha \in \mathcal{T}.S \preceq \alpha.x :: Dim(dom, \alpha) \text{where} dom = \bigcup dom_{E_j}} \text{(T—DefDimPartial)}$$

Function and Operator Declarations and Definitions

$$\frac{\begin{array}{c} f : Function(T_{1..n}, R).\{A_{1..m}\} \in \Gamma \qquad \Gamma \vdash T_{1..n}, R \prec \mathcal{T} \\ \mathcal{A} \vdash Function(T_{1..n}, R).\{A_{1..m}\} \preceq A_{1..m}.target \end{array}}{\Gamma \vdash f :: Function(T_{1..n}, R).\{A_{1..m}\} \qquad \exists \mathcal{F}_f} \text{(T—FuncDecl)}$$

$$\frac{f(t_{1..n} : S_{1..n}) = \lambda(t_{1..n} : S_{1..n}).E \in \Gamma \qquad \Gamma \vdash f : Function(T_{1..n}, R) \qquad \Gamma_E \vdash E : R' \preceq R \qquad (S_{1..n} \preceq T_{1..n})}{\mathcal{F}f \vdash f_{S_{1..n},R'}(t_{1..n} :: S_{1..n}) = \lambda(t_{1..n} : S_{1..n}).E \qquad \Gamma_E \vdash t_i :: S_i} \quad \text{(T—FuncDef)}$$

$$\frac{\phi : Operator(T_{1..n}, R).\{A_{1..m}\} \in \Gamma \qquad \Gamma \vdash T_{1..n}, R \in \mathcal{T} \qquad \mathcal{A} \vdash Operator(T_{1..n}, R).\{A_{1..m}\} \preceq A_{1..m}.target}{\Gamma \vdash \phi : Operator(T_{1..n}, R).\{A_{1..m}\} \qquad \exists \mathcal{O}\phi} \quad \text{(T—OpDecl)}$$

$$\frac{\Gamma \vdash \phi : Operator(T_{1..n}, R) \qquad \phi(t_{1..n} : S_{1..n}) = \lambda(t_{1..n} : S_{1..n}).E \in \Gamma_\phi \qquad \Gamma_\phi \vdash E : R' \preceq R \qquad S_{1..n} \preceq T_{1..n}}{\mathcal{O}\phi \vdash \phi_{(S_{1..n},R'}(t_{1..n} : S_{1..n}) = \lambda(t_{1..n} : S_{1..n}).E} \quad \text{(T—OpDef)}$$

A.2.9 Expressions

Literals

$$\frac{c \in \{Literal_{Integer}\}}{\Gamma \vdash c : IntegerRange([c..c])} \quad \text{(T—LitInt)}$$

$$\frac{c \in \{Literal_{Real}\}}{\Gamma \vdash c : IntegerRange([c..c])} \quad \text{(T—LitReal)}$$

$$\frac{c = Tuple(t_{1..n}) \in \Gamma_E \qquad \Gamma_E \vdash t_{1..n} : T_{1..n}}{\Gamma \vdash c : Tuple(T_{1..n})} \quad \text{(T—LitTuple)}$$

$$\frac{c = Record(l_{1..n} = t_{1..n}) \in \Gamma_E \qquad \Gamma_E \vdash t_{1..n} : T_{1..n}}{\Gamma \vdash c : Record(l_{1..n} : T_{1..n})} \quad \text{(T—LitRecord)}$$

$$\frac{c = [t_{1..n}] \qquad \Gamma \vdash t_{1..n} : T_{1..n}}{\Gamma \vdash c : Array([1..n], T) \qquad \Gamma \vdash T = \sqcup T_{1..n}} \quad \text{(T—LitArray)}$$

Variables

$$\frac{\Gamma \vdash x :: T \qquad \Gamma \vdash y = \lambda x.E_2 : R}{\Gamma \vdash \{\forall E_1 : S \preceq T\} y = [x \mapsto E_1]E_2 : R} \quad \text{(T—VarEval)}$$

$$\frac{\Gamma \vdash x = E \qquad \Gamma \vdash x : T \qquad \Gamma_E \vdash E : S}{\Gamma \vdash S \preceq T} \qquad \text{(T—DefInfer)}$$

$$\frac{\Gamma \vdash x :: T \qquad \text{Sh}(T) = \mathit{Record}(l_{1..n} : T_{1..n})}{\Gamma \vdash \exists S_{1..n} \in \mathcal{T}.S_{1..n} \preceq T_{1..n}.x.l_{1..n} :: S_{1..n}} \qquad \text{(T—RecField)}$$

$$\frac{\Gamma \vdash x : T \qquad \text{Sh}(T) = \mathit{Tuple}(T_{1..n}) \qquad \Gamma \vdash x = (x_{1..n})}{\Gamma \vdash \exists S_{1..n} \in \mathcal{T}.S_{1..n} \preceq T_{1..n}.x_{1..n} :: S_{1..n}} \text{(T—TupleField)}$$

$$\frac{\Gamma_E \vdash x_{1..n} : T_{1..n}}{\Gamma \vdash (x_{1..n}) : \mathit{Tuple}(T_{1..n})} \qquad \text{(T—DefTupleConst)}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : T \qquad \text{Sh}(T) = \mathit{Array}(\mathit{dom}, \alpha) \qquad x[i_1, ..., i_n] \in \Gamma_E \\ \exists l_{1..n}, u_{1..n} \in \mathcal{Z} \ni \mathit{dom} = \mathit{bigcross}_{([l_{1..n}..u_{1..n}])}\end{array}}{\Gamma_E \vdash \mathit{ind}_j : S_i \preceq \mathit{IntegerRange}[l_j..u_j] \qquad \Gamma_E \vdash \exists \beta \in \mathcal{T}.\beta \preceq \alpha.\{\forall(\mathit{ind}_{1..n}) \in \mathit{dom}\}.x[i_1, ..., i_n] :: \beta}$$
(T—ArrayIndex)

Function Implementations and Function Applications (Joins)

$$\frac{\begin{array}{c}\Gamma \vdash x = f_{T_{1..n},R}(t_{1..n}) \qquad \mathcal{F}_f \vdash f_{T_{1..n},R} = \lambda p_{1..n} : T_{1..n}.E_{T_{1..n},R} \\ \Gamma_x \vdash t_{1..n} : S_{1..n} \qquad S_{1..n} \preceq T_{1..n}\end{array}}{\Gamma, x : R' \vdash R' \preceq R}$$
(T—FuncImplement)

$$\frac{\begin{array}{c}\Gamma_E \vdash y = f(t_{1..n}) \qquad \Gamma_E \vdash t_{1..n} : S_{1..n} \qquad \Gamma \vdash f : \mathit{Function}(T_{1..n}, R) \qquad S_{1..n} \preceq T_{1..n} \\ \Gamma \vdash f(t_{1..n}) \preceq Q_{1..m} \qquad \bot_T \prec \sqcap Q_{1..m}\end{array}}{\Gamma \vdash y : R* \qquad R* \preceq \sqcup\{\forall R' \ni f_{S'_{1..n},R'} \in \mathcal{F}f \wedge S_{1..n} \preceq S'_{1..n} \wedge R' \preceq \sqcap Q_{1..m}\}.R'}$$
(T—FuncJoin)

$$\frac{\begin{array}{c}x = \phi(t_{1..n}) \in \Gamma_E \qquad \Gamma_E \vdash t_{1..n} : S_{1..n} \qquad \Gamma \vdash \phi \in \mathit{Operators} \\ \Gamma \vdash x \preceq Q_{1..m} \qquad \bot_T \prec \sqcap Q_{1..m}\end{array}}{\Gamma \vdash x : R* \qquad R* \preceq \sqcup\{\forall R' \ni \mathit{op}_{(T_{1..n}),R'} \in \mathcal{O}\phi \wedge S_{1..n} \preceq T_{1..n} \wedge R' \preceq \sqcap Q_{1..m}\}.R'}$$
(T—OpJoin)

A.3 Proof Obligations

This section covers the formal semantics for proof obligations, run time tests, assertions and attribute propagation rules. An informal description is provided in 5.7.

In the rules below, the following notations are used:

$\mathcal{P}$ the set of outstanding proof obligations for the module

$\mathcal{R}_{\mathcal{T}}$ the set of run time test obligations for attributes whose class defines such a test

$$\mathcal{P} = \emptyset \mid \mathcal{P}, (E, A(\rho)) \qquad \text{(C—ProofObligations)}$$

$$\frac{\Gamma_E \vdash E :: T \qquad E' = \models (E, A^T(\rho)) \in \Gamma \qquad T \neq T.A^T(\rho) \qquad \mathcal{A} \vdash T \preceq A^T.target \qquad \mathcal{A} \vdash A(\rho) \prec^A A(E) \qquad \models :: A.target \rightarrow \mathcal{B} \in \mathcal{A}}{(E, A(\rho)) \in \mathcal{R}_{\mathcal{T}} \qquad \Gamma \vdash E' : T.A(\rho)} \text{(A—Test)}$$

$$\frac{\Gamma_E \vdash E :: T \qquad E' = \models (E, A^T(\rho)) \in \Gamma \qquad \mathcal{A} \vdash T \preceq A.target \qquad (T = T.A(\rho)) \vee \mathcal{A} \vdash A^T(E) \preceq^A A^T(\rho)}{\Gamma \vdash E' : T.A(\rho)}$$

(A—RedundantTest)

$$\frac{\Gamma_E \vdash E : T \qquad E' = Assert(E, A^T(\rho)) \in \Gamma \qquad T \neq T.A^T(\rho) \qquad \mathcal{A} \vdash T \preceq A^T.target \qquad \mathcal{A} \vdash A^T(\rho) \prec^A A^T(E)}{(E, A(\rho)) \in \mathcal{P} \qquad \Gamma \vdash E' : T.A^T(\rho)} \text{(A—Assert)}$$

$$\frac{\Gamma_E \vdash E : T \qquad E' = Assert(E, A^T(\rho)) \in \Gamma \qquad \mathcal{A} \vdash T \preceq A^T.target \qquad (T = T.A^T(\rho)) \vee \mathcal{A} \vdash A^T(E) \preceq^A A^T(\rho)}{\Gamma \vdash E' : T.A(\rho)}$$

(A—RedundantAssert)

$$\frac{x :: T \in \Gamma \qquad \mathrm{Attr}(T) = \{A^T_{1..n}(\rho_{1..n})\} \qquad \mathcal{A} \vdash T \preceq A^T_{1..n}.target \qquad \rho_{1..n} \preceq A^T_{1..n}.property \qquad x \in \Gamma_{In}}{\mathcal{P} \vdash \{A^T_{1..n}(\rho_{1..n})\}} \text{(A—VarDeclInput)}$$

$$\frac{\begin{array}{c} x :: T \in \Gamma \qquad \text{Attr}(T) = \{A^T_{1..n}(\rho_{1..n})\} \\ \mathcal{A} \vdash T \preceq A^T_{1..n}.target \qquad \rho_{1..n} \preceq A^T_{1..n}.property \\ \Gamma \vdash x = E \qquad \Gamma \vdash E :: S, \text{Attr}(S) = \{A^T_{1..k}(\mu_{1..k}), A^S_{k+1..m}\} \end{array}}{\mathcal{P} \vdash \forall (A^T_i(\mu_i) \npreceq^A A^T_i(\rho_i))_{1..k}.\{A^T_i(\rho_i)\}} \text{(A—VarDeclDef)}$$

$$\frac{\begin{array}{c} \Gamma \vdash y = \sqcup\{\forall S'_{1..n}, R' \ni f_{S'_{1..n},R'} \in \mathcal{F}f \wedge S \preceq S' \wedge R' \preceq \sqcap Q_{1..m}\}.f_{S'_{1..n},R'}(t_{1..n}) \\ \mathcal{F}f \vdash f_{S,R'}(t_{1..n}) = \lambda(t_{1..n}).E \qquad \Gamma_E \vdash E : S \\ A^{R'}_j(\rho_j) \in \text{Attr}(R') \qquad A^{R'}_j(S) \npreceq^A \rho_j \end{array}}{(f_{S_{1..n},R'}, A^{R'}_j(\rho_j)) \in \mathcal{P}}$$

(A—FuncJoin)

$$\frac{\begin{array}{c} \Gamma \vdash x = f_{T,R}(t_{1..n}) \qquad \mathcal{F}_f \vdash f_{T,R} = \lambda p_{1..n} : T_{1..n}.E_{T_{1..n},R} \qquad \Gamma_x \vdash t_{1..n} : S_{1..n}, S_{1..n} \preceq T_{1..n} \\ context, x : R' \vdash R' \preceq Q_{1..m} \qquad \bot_T \prec Q = \sqcap Q_{1..m} \\ (A^{Q_{1..m}} \in \text{Attr}(Q) \wedge A^{Q_{1..m}} \notin \text{Attr}(R)) \qquad \mathcal{A} \vdash A^{Q_{1..m}}((E_{T,R})(t_{1..n}) = \rho \end{array}}{\Gamma, x : R' \vdash R' \preceq R.\{A^{S_{1..n}}(\rho)\}}$$

(A—FuncInferRes)

$$\frac{\begin{array}{c} \Gamma \vdash x = f_{T,R}(t_{1..n}) \qquad \mathcal{F}_f \vdash f_{T,R} = \lambda p_{1..n} : T_{1..n}.E_{T_{1..n},R} \\ \Gamma_x \vdash t_i : S_i \qquad S_i \preceq T_i,\ 1 \leq i \leq n \\ A^{S_j} \in \text{Attr}(S_j) \wedge A^{S_j} \notin \text{Attr}(T_j) \qquad \mathcal{A} \vdash A^{S_j}((E_{T,R})(t_{1..n})) = \rho \end{array}}{\Gamma, x : R' \vdash R' \preceq R.\{A^{S_j}(\rho)\}}$$

(A—FuncPropArg)

$$\frac{\begin{array}{c} \Gamma \vdash y = \sqcup\{\forall (\phi, R') \in \mathcal{O}\phi \ni S_i \preceq T_i \wedge R' \preceq \sqcap Q_{1..m}\}.\phi_{T_i,R'}(t_{1..n}) \\ \mathcal{O}_\phi \vdash \phi_{T_{1..n},R'} = \lambda p_{1..n}.E_{T,R'} \qquad \Gamma_{E'_{T,R}} \vdash E_{T,R'} :: S \\ A^{R'}_j(\rho_j) \in \text{Attr}(R') \qquad A^{R'}_j(S) \npreceq^A \rho_j \end{array}}{(\phi_{(T_1,...,T_n),R'}, A^{R'}_j(\rho_j)) \in \mathcal{P}} \text{(A—OpJoin)}$$

$$\frac{\begin{array}{c} \Gamma \vdash x = \phi_{(T_1,...,T_n),R}(t_1, ..., t_n) \qquad \mathcal{O}_\phi \vdash \phi_{(T_1,...,T_n),R} = \lambda p_1 : T_i, ..., p_n : T_n.E_{T,R} \\ \Gamma_x \vdash t_i : S_i \qquad S_i \preceq T_i,\ 1 \leq i \leq n \\ context, x : R' \vdash R' \preceq Q_i,\ 1 \leq i \leq n \qquad \bot_T \prec Q = \sqcap Q_i \\ (A^{Q_j} \in \text{Attr}(Q) \wedge A^{Q_j} \notin \text{Attr}(R)) \qquad \mathcal{A} \vdash A^{Q_j}((E_{T,R})(t_1, ..., t_n)) = \rho \end{array}}{\Gamma, x : R' \vdash R' \preceq R.\{A^{S_j}(\rho)\}}$$

(A—OpInferRes)

$$\frac{\Gamma \vdash x = \phi_{(T_{1..n}),R}(t_1, ..., t_n) \qquad \mathcal{O}_f \vdash \phi_{(T_{1..n}),R} = \lambda p_{1..n} : T_{1..n}.E_{T,R} \qquad \Gamma_x \vdash t_i : S_i \qquad S_i \preceq T_i,\ 1 \leq i \leq n \qquad A^{S_j} \in \text{Attr}(S_j) \wedge A^{S_j} \notin \text{Attr}(T_j) \qquad \mathcal{A} \vdash A^{S_j}((E_{T,R})(t_1, ..., t_n)) = \rho}{\Gamma, x : R' \vdash R' \preceq R.\{A^{S_j}(\rho)\}}$$

(A—OpPropArg)

Appendix B

Concrete Syntax eBNF

This is a faithful description of the language as parsed by the HUSC system in eBNF. A few of the features have not been implemented, and these are clearly labelled here.

B.0.1 Lexical Structure

The HUSC lexicon is based on Haskell, but with some effort made to make the source code compatible with Tex / Latex processing for future work. For example, backslashes have been excluded to support future Tex labelling of non-ASCII chacters.

A large number of reserved words are used in this concrete syntax, with the intent of making the code more like written mathematics. This has the downside of making the language quite verbose, but the modules should generally be quite small and context heavy. It remains to be seen whether this strategy will be successful, or just annoying.

```
module Lex where

import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as P
-- import Text.ParserCombinators.Parsec.Expr as ParsecExpr
import Text.ParserCombinators.Parsec.Language

tokparse        = P.makeTokenParser twoCoconutDef
```

```
lexeme          = P.lexeme tokparse
whiteSpace      = P.whiteSpace tokparse
symbol          = P.symbol tokparse
operator        = P.operator tokparse
reserved        = P.reserved tokparse
reservedOp      = P.reservedOp tokparse

semiSep         = P.semiSep tokparse
semiSep1        = P.semiSep1 tokparse
commaSep        = P.commaSep tokparse
commaSep1       = P.commaSep1 tokparse
semi            = P.semi tokparse
colon           = P.colon tokparse
dot             = P.dot tokparse

braces         = P.braces tokparse
parens         = P.parens tokparse
squares        = P.squares tokparse
angles         = P.angles tokparse
natural         = P.natural tokparse
decimal         = P.decimal tokparse
hexadecimal     = P.hexadecimal tokparse
octal           = P.octal tokparse
integer         = P.integer tokparse
float           = P.float tokparse
naturalOrFloat  = P.naturalOrFloat tokparse
charLiteral     = P.charLiteral tokparse
stringLiteral   = P.stringLiteral tokparse

-- identifier   = P.identifier tokparse    can't use, need to differentiate no trailing

identifier       = do {s <- letter; ss <- many (alphaNum <|> oneOf "_'"); return $ s:ss}
name             = do{ n<-identifier; whiteSpace; return n}
```

```
twoCoconutDef = haskellStyle {
   opStart = oneOf "`~!#%^&*:/?-+[]<>.=",  -- no backslash, no @ allowed
   opLetter = oneOf "`~!#%^&*:/?-+[]=.",
   identStart = letter,
   identLetter = alphaNum <|> oneOf "_'",
   reservedOpNames = ["=", "::", ",", "|" ],
   reservedNames =
     ["Let", "let", "be", "Define", "define", "as", "of",
      "from", "to", "in", "where", "if", "is", "are", "on",
      "Frame", "Module",  "export","input", "Use", "use", "import", "with", "qualified",
      "join", "Join", "select", "Select", "Otherwise", "otherwise", "For", "for", "all",
      "Asserting","asserting", "Testing","testing",
      "Type", "based", "True", "False",
      "And", "and", "or", "Or"
     ],
   caseSensitive  = True
   }

-- end module Lex
```

B.0.2 BNF Terminals / Tokens

In the current implementation, there is no lexical distinction between different types of identifiers. The recommended strategy is to give module type identifiers uppercase names, and data identifiers with lower case names, but this is not enforced.

The prefix “coco” is reserved for all identifiers, and the symbol is also reserved for internal use.

Identifier	any letter, then letter, number or _
OpSymbol	any sequence of operator characters
ModuleId	an identifier, recommended to begin with upper case letter
TypeId	an identifier, recommended to begin with uppercase letter
TypeVar	type identifier.
Label	string applied to body of a function/operator definition for documentation
Annotation	place holder for passing information to lower levels of Coconut, like a pragma

Unless otherwise specified (*nospace*), tokens are assumed to be separated by whitespace.

The lexer assumes that the source code has already been extracted from a literal source file.

B.0.3 Module Headers

`Module`	→	"Module" *ModuleId* [`InputDecl`]? `ExportDecl` `ModuleBody`
`InputDecl`	→	"import" `DeclList`
`ExportDecl`	→	"export" [[`bnftrmIdentifier` | *TypeId* | *OpSymbol*]]*
`ModuleBody`	→	'{' [`ModuleImport` | `Declaration` | `TypeDef` | `Definition`]* '}'

B.0.4 Declarations

`DeclList`	→	`DeclCompact` [','`DeclCompact`]*
`Declaration`	→	[`DeclCompact` | `DeclBaroque`]
`DeclCompact`	→	`IdList` ["is" | "::"] `FullType`
`DeclBaroque`	→	"Let" `IdList` "be" `FullType`
`IdList`	→	*Identifier* ["," *Identifier*]*

B.0.5 Module Imports

`ModuleImport`	→	["Use" | "use"] *ModuleId* ["`qualified`"]? ["`as`"*ModuleId*]?
		["`with`"`DefinitionList`]?
		["`import`"]? `DefinitionList`
`DefinitionList`	→	'{' `Definition` [','`Definition`]*'}'

B.0.6 Data Definitions

`Definition`	→	[`QuantExpr`]? *Identifier nospace* [`FunctionDef`|`PartialDef`]?
		'=' `JoinExpr` [`LocalBody`]?
`FuncDef`	→	'(' [`ParmVar`[','`ParmVar`]*]? ')'
`PartialDef`	→	[`PartialArrayDef` | `PartialRecordDef`]
`PartialArrayDef`	→	'[' `IndexSpec` [','`IndexSpec`]* ']'
`PartialRecordDef`	→	'.' `FieldSpec`
`ParmVar`	→	[*Identifier* | `DeclCompact`]
`IndexSpec`	→	[`Identifier` | `Expr`]
`IdentField`	→	*Identifiernospace*[`FunctionDef`|`PartialDef`]?
`QuantExpr`	→	["For" | "for"] `QuantExprItem` [','`QuantExprItem`]*
`QuantExprItem`	→	["all" *Identifier* | `Definition`]
`LocalBody`	→	"where" [*Label*]? '{' [[`Declaration` | `Definition`]]+ '}'

B.0.7 Operator Definitions

`OperatorDefinition`	→	`OperatorInterface` [`OperatorImplement`]?
`OperatorInterface`	→	['('`DeclList`')']? *OperatorSymbol* ['('`DeclList`')']?
		" = " `Declaration`
`OperatorImplementation`	→	`LocalBody`

B.0.8 Joins and Conditional Expressions

JoinExpr	→	"Join" '['JoinAnnotExpr [';' JoinAnnotExpr]*']' | CondExpr
JoinAnnotExpr	→	CondExpr JoinAnnotation
JoinAnnotation	→	*Annotation*
CondExpr	→	[AnnotatedExpr | "Select" '{' CondExprItem [';' CondExprItem]* '}']
CondExprItem	→	AnnotatedExpr | BoolGuard
BoolGuard	→	[BoolExpr | *otherwise*]
BoolExpr	→	Expr
AnnotatedExpr	→	[["::" FullType] | [[Assertions]? [RunTimeTests]?]]
AddAttributes	→	[[Assertions | RuntimeTests]]*
Assertions	→	"asserting" AttributeList
RuntimeTests	→	"testing" AttributeList

B.0.9 Operators in Expressions

Expr → Factor
 | Expr InfixOp Expr
 | PrefixOp Expr
 | Expr PostfixOp

InfixOp	→	*OpSymbol*
PrefixOp	→	*OpSymbol*
PostfixOp	→	*OpSymbol*

B.0.10 Factors

Factor	→	Literal | Variable | '(' Expr ')'
Variable	→	Identifier [*nospace*IndexExpr | ParmsExpr | FieldLabel]?
IndexExpr	→	'[' Expr [',' bnfntExpr]? ']'
ParmsExpr	→	'(' Expr [',' bnfntExpr]? ')'
FieldLabel	→	'.' Variable
Literal	→	*LitStr* | *LitInt* | *LitReal*

B.0.11 Types and Type Definitions

`TypeDef`	→	"Define" `TypeId` [',' `TypeId`]* [`TypeParms`]?
		[["as" | "based" "on"]]? `FullType`
`FullType`	→	`GenericType` | `NamedType`
`NamedType`	→	*TypeId* [`DepArgs`]? `FullType` `AttributeList`
`GenericType`	→	*TypeId* `AttributeList` `Shape`
`Shape`	→	*Scalar*
		Record '(' `Declaration` [',' `Declaration`]* ')'
		Tuple '(' `FullType` [',' `FullType`]* ')'
		DimType nospace '[' `IntExpr` [',' `IntExpr`]* ']' "of" `FullType`
		FuncType "from" '(' `FullType` [',' `FullType`]* ')' "to" `FullType`
`AttributeList`	→	'' `Attribute` [',' `Attribute`]* ''
`Attribute`	→	*AttrClassId nospace* '(' `AttrProperty` ')'
`DepArgs`	→	'(' `Expr` [',' `Expr`]* ')'

B.0.12 Attribute Classes

This syntax has not been implemented in the current implementation, and likely will not be as it did not seem to be sufficient to define the propagation rules. It has been provided to illustrate what makes up an attribute class.

The special function and predicate definitions for an attribute class are identified by having a special name incorporating a prefix and the attribute class name (e.g isSymmetric, subRange). Otherwise they are syntactically just definitions.

`AttributeClass`	→	"Attribute Class" *AttrClassId* '(' `AttrProperty` ')'
		"of" `FullType` "where" '{'
		[[`SpecialAttr` | `Declaration` | `Definition`]* '}'
`AttrProperty`	→	`FullType`
`SpecialAttr`	→	`IsAttr` | `SubAttr` | `AttrRule`
`IsAttr`	→	`Definition`
`SubAttr`	→	`Definition`
`AttrRule`	→	`Definition`

Appendix C

Implementation

There are two fundamental components to the system, the HUSC parser and the type system. The HUSC parser is written in (literate) Haskell, using the Parsec combinators, and records a HUSC source file as a HUSC program, a main module with a set (map) of modules that are imported (right now all modules must be in the same source file, until a decision is made on how the module directory will be maintained). The type system is written in CHR and SWI Prolog, with the module definitions being encoded as Prolog relations and / or CHR constraints or rules to carry out the type inferencing. The output of the type system can then be imported back into Haskell, and matched with the definitions to produce the program hypergraph. *This has not been implemented, but the rules for graph generation have been included*

The following steps process a literate HUSC file into the output of the type system. Following the list of steps is a detailed description, including the literate Haskell source and the (sadly illiterate) CHR/Prolog code for each of the modules.

1. Run "coco2" from the command line, enter the name of the literate HUSC file (*name*.nut) to be processed, where *name* can be any label but by convention should be the main module name. Currently, the first module is assumed to be the main module, and all of the modules to be imported must be in the same file (this is obviously inappropriate long term).

2. The main module uses CocoUnlit to create a pure (deliterated) HUSC file, which is written out as *name*.nutty, including line numbers. Parser errors will refer

to these line numbers, not the line numbers from the literate file. The parser should be rewritten to provide the line numbers from the .nut files, but in the mean time the .nutty file is useful for solving parser errors.

3. The CocoParser module is invoke to translate the HUSC source into internal data structures, with each statement becoming a definition. The definitions are split out by class (e.g. data definition, type definition) and stored in individual lists (maps) within the module.

4. The main program then invokes the CocoSyntax module to test for structural errors in the modules. For example, local bodies of expressions (functions, operators, etc.) may only contain variable declarations and definitions, not type, operator or attribute class definitions. Errors are recorded as part of the module data structure by identifier (mapped). If there are any syntax errors, the program will not be encoded for the type system.

5. The main module invokes CocoTypes to translate the main module, including it's imported definitions, from Haskell into two CHR/Prolog source files called *name*_defs and *name*_types.

6. Invoking the SWI Prolog compiler for constraint.pl, with the goal "go(*modname*)", in the same directory as these two files, will execute the type system. This can be from the interactive window

```
swipl

[constraint]

go({\it modname})
{\it results follow...}
```

or from the command line, optionally piping the output into a file:

```
swipl -s constraint.pl -g ''go({\it modname})`` -t halt >> tsout.txt
```

> *The results are currently dumped to stdout, but should really be written to (at least) two files, one for important type information and errors, the other storing extra information for trace / debugging.*

At this point in time, there are no specific requirements for the output of the HUSC compiler front end. It is reasonably straight forward to extract the results of the type system and merge it with the Haskell structures to form a code hypergraph. It is a hypergraph because each operation, function application and explicit join represents alternate, equivalent edges between two data term nodes. The type system produces some number of valid interface choices for each such junction, possibly zero, which then forms the "join expression" or hypergraph edge. No valid choices for the expression is considered an error.

C.1 HUSC Programs and Modules

The main Coconut module handles parsing and type checking and inferencing. The input is a set of literate HUSC source module (in a .nut file), which is then processed to extract pure HUSC source into a .nutty file. This pure source file is parsed, checked for completeness and run through the type system. The output is a hypergraph with type annotations, which is written to a file with extension .milk.

The following steps occur:

deliterate the CocoUnlit module extracts the HUSC code from a literate source file. Coconut code is either in a Latex "cococode" environment or a single line is prefixed by "coco¿". The HUSC code is written to an intermediate file with a .nutty extension, which can then be used when checking the line number of parser errors.

parsing the source modules in the file are parsed into CocoModule structures.

completeness checks takes the first module in the file as the main program, then resolves all module imports into that module, in a depth first approach. When a module is imported, it's identifiers are renamed to simulate scoping. Finally,

type and data definitions are checked to make sure all definitions are formed of well defined variables, input definitions or literals.

type evaluation the internal definitions of the main module, including it's imports, are translated into two files for import by the type system. The *module*_types.pl file contains type definition Prolog predicates, the *module*_defs.pl file contains data declarations and definitions as CHR constraints (in a CHR/Prolog format). Running SWI prolog with main file constraint.pl, goal "run", will complete the type system evaluation.

C.1.1 Data Structures

This is a quick overview of the data structures that make up the modules.

CocoModule data and type definitions, operator symbols, of a module.

TypeMap / TypeDef a type definition defines a new type, as a subtype of or based on another type. The TypeMap is just a collection (map) of types, keyed on the type name.

AttrClass provides the definitions of an attribute class (*this has not been implemented, see earlier discussion for details, only Scalar and Range attributes are available*).

Data DataDef or DataDecl, a data identifier definition or type declaration. Data definitions contain an identifer, Def structure, local body and quantified variables where appropriate.

OperDef associates a function definition with an operator symbol; operators can be binary, unary preop or postop.

Def defines a complete, partial or a function definition as part of a data or operator definition. This includes a join expression, the domain of a partial definition as a set of integer ranges, or the parameters and result type of a function or operator definition.

FullType describes an instance of a type (as opposed to a type definition) and makes up part of a definition.

Attribute records a specific attribute instance (as opposed to an attribute class definition) consisting of an attribute class and an attribute property value.

JoinExpr is a list of conditional expressions that are joined, with their guards.

CondExpr is a list of expressions and the guards in the form of attribute assertions or tests.

Expr an expression of BinOps, UnOps, Vars (including function applications and array elements), Lits and joined expressions in parentheses.

Quantifier describes the integer domain of a universally quantified variable in a partial definition, either "for all x" or "for $x =$ *integer sequence*".

LocalBody contains local Data definitions, scoped to a particular definition.

.

Issues and Future Work

All modules must be in the same source file, this is a temporary step until a system for locating modules is determined.

The results from the type system are not merged with the data structures and definitions in the module. The plan is to create a hypergraph merging the two sets of information, but without a specific set of requirements there is no point in completing this at this time.

C.1.2 Main HUSC Program

```
--import Data.Map hiding (map)
import IO
import qualified Data.Map as Map


-- coconut modules
```

```
import CocoUnlit
import HUSCModule
import EncTypes  (encMod)

main :: IO ()
main = do { putStr "HUSC source file (.nut): "
          ; hFlush stdout
          ; infile <- getLine;
          ; putStrLn ""
          ; coco infile
          }

coco :: String -> IO ()
coco infile
    = do { codefile <- unLitFile infile
         ; let code = cocoParse codefile
         ; case (code) of
             Left literr -> putStrLn ("parse error at " ++ (show literr))
             Right parsedProg ->
               do { let synProg = chkProgSyntax parsedProg
--                  ; dspCocoProg synProg "Checked"
--                  ; print synProg
                  ; let liftProg = mergeMods synProg
                  ; dspCocoProg liftProg "Merged"
                  ; cocoTypeOut liftProg
                  }
         }


cocoTypeOut :: CocoProgram->IO ()
cocoTypeOut (CocoProgram mainmod impmods) =
  do { let prgname = modName mainmod
     ; let deff = (defPathFromMod prgname)
     ; let typef = (typePathFromMod prgname)
     ; let outpath = (outPathFromMod prgname)
```

```
     ; writeFile  deff ("%% HUSC program "++prgname++",v0.1\n\n")
     ; appendFile deff ("run(M) :- ")
     ; let rels = (encMod mainmod)++(concat (map encMod (Map.elems impmods)))
     ; mapM_ (\s-> appendFile deff ("\n    "++(s++","))) (filter forGoal (makeLine rels)
     ; appendFile deff ("\n    cleanup('"++outpath++"').\n")
     ; writeFile  typef (":- module('"++prgname++"_types',[type/4,funcintf/5,funclass/4]
     ; mapM_ (\s-> appendFile typef ('\n':(s++"."))) (filter forTypes (makeLine rels))
     ; appendFile typef "\n"
     }

-- concatenates relations to fill a line
lmax = 100
makeLine :: [String]->[String]
makeLine [] = []
makeLine (s:[]) = [s]
makeLine (s:ss)
    | ((length s) > lmax) = (s) : (makeLine ss)
    | (forTypes s) = (s) : (makeLine ss)
makeLine (s:(s2:ss))
    | (length (s++s2) < lmax) = makeLine ((s++","++s2):ss)
    | otherwise               = (s) : (makeLine (s2:ss))

forGoal,forTypes:: String->Bool
forTypes = (\s -> let pred = takeWhile ('(' /=) s in elem pred typePreds)
forGoal =  not.forTypes
typePreds = ["type","funcintf","funclass"]

defPathFromMod mod = "CHRs/"++mod++"_defs.pl"
typePathFromMod mod = "CHRs/"++mod++"_types.pl"
outPathFromMod mod = "CHRs/"++mod
```

C.2 HUSC Modules

A module is a named collection of data and type definitions, plus an operator symbol table. The module exports a subset of it's definitions, specified by a list of data and type identifiers (and operator symbols, all of the operator interfaces and implementations associated with the specified symbol will be exported).

Modules may also require definitions from the importing module, specifying the identifier and a type declaration. Imported definitions must satisfy the stated type constraints. In this way importing a HUSC module involves a bidirectional exchange of definitions, the importing module must satisfy the requirements of the imported module.

The opTable is a table of recognized operator symbols, along with the associativity and precedence rules for those symbols. The operator symbol table is currently hard coded into the CocoParser module, and is based on the default table for Parsec. The architecture of the language calls for it to be possible to define new symbols, with a precedence and associativity, to be added to the opTable, but this has not been implemented yet.

```
module HUSCModule where

import qualified Data.Map as Map
import qualified List(partition)

import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as P
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Perm

import MiscST
import Lex
import VarSub as Sub
import Expr
import Types
import JoinExpr
```

```
import DataDefs
import OperDef
import TypeDef
import AttrClass

data ModuleStatus = Raw | Parsed | Checked | Merged | Typed
    deriving (Show, Eq)

-- other map synonyms given in the specialized modules.
type ModuleMap = IdMap CocoModule
type ErrorMap = IdMap String

data CocoProgram = CocoProgram {mainMod::CocoModule, impMods:: ModuleMap}
  deriving Show

data CocoModule = CocoModule
  { modName    :: Identifier
  , modExports :: [IdSym]
  , modStatus  :: ModuleStatus
  , modOpTable :: OpSymbolTable
  , modOpDefs  :: [OperDef]
  , modDefs    :: [Data]
  , modTypes   :: TypeMap
  , modACs     :: [AttrClass]
  , modImps    :: [ImportDef]
  , modErrors  :: ErrorMap
  }

instance Show CocoModule where
  showsPrec p (CocoModule n exps sts ops opdefs defs tts acs imps errs) =
      ("CocoModule("++).(n++).(", status:"++).(showsPrec p sts).
      (", exports :"++).(showsPrec p exps).
      ("\ntype defs:\n"++).(showsPrec p (Map.elems tts)).
      ("\ndefinitions:\n"++).(showsPrec p defs).
```

```
      ("\noperator defs:\n"++).(showsPrec p opdefs).
      ("\nusing \n"++).(showsPrec p imps).
      ("\n  errors"++).(showsPrec p errs)

-- build new substitution for module by
-- rename data and type definitions
instance AllowSubs CocoModule where
  substitute sub scope (CocoModule id exps st ops opdefs defs tts acs imps errs) =
       CocoModule (Sub.apply sub id) (substitute sub scope exps) Merged ops
             ( substitute sub scope opdefs)
             (substitute sub scope defs)
             (treeSub substitute sub scope tts)
             ( substitute sub scope acs)
             ( substitute sub scope imps)
              errs
  rename sub scope (CocoModule id exps st ops opdefs defs tts acs imps errs) =
    let (lsub1, newtts) = liftTypeTree sub scope tts in
      let (lsub2, newdefs) = liftVars (lsub1 `Map.union` sub) scope defs in
        let newsub = lsub1 `Map.union` lsub2 `Map.union` sub in
          let newmod =
                CocoModule id (substitute newsub scope exps) st ops
                  (snd (rename newsub scope opdefs))
                  newdefs
                  newtts
                  acs
                  (substitute newsub scope imps) errs in
            (newsub, newmod)
```

C.2.1 Parsing Modules

Parsing a module generates a CocoModule structure. The same module data structure is used over a number of stages, so the status of the module is represented in the modStatus field.

The parser developed here is the character based, stateless parser written using Parsec combinators as shipped with GHC 6.4.1. This module parses pure HUSC source files, meaning the literate components have already been removed. This proves to be problematic because it is difficult to match the parse errors produced from the pure source with the literate source file, the line and column numbers being offset. In addition, the operator table in this version is static, instead of being updated within the modules.

The function cocoParse treats the unparsed file as a string, and returns either a CocoModule structure, consisting of definition blocks (DefBlocks), or a parse error. The input file is assumed to be deliterated (unliterated? illiterate?); the CocoUnlit module takes care of this. More than one module may exist in the same input file.

A module begins with the module name preceded by the label module: This is followed first by a list of output identifiers (or operator symbols), with at least one export definition. An optional list of input variable definitions, identifies those definitions that must be provided by any module that imports this module. Any other definitions within the module are private; there is currently no ”export all” option.

Note: the terms import / input and export / output have been used somewhat interchangeably in the parser. This is because there was some debate about which was most applicable; in the end the module was determined to input definitions from it’s importer, output definitions to the module importing it, and import definitions from other modules used internally. This needs to be resolved in the code.

```
cocoParse :: String -> Either ParseError CocoProgram
cocoParse codefile = parse (cocoProgram) "" codefile

cocoProgram :: Parser CocoProgram
cocoProgram =
  do { mm <- cocoModule
     ; ms <- many cocoModule
     ; return $ CocoProgram mm (Map.fromList [((modName m),m) | m<-ms])
     }

cocoModule :: Parser CocoModule
```

```
cocoModule
     = do { whiteSpace;
            reserved "module";
            n   <- name;
            el  <- export_ids;
            il  <- option [] import_decl_vars;
            sts <- codeblock;
            return $ mkModule n el il sts
          }

export_ids :: Parser [IdSym]
export_ids = do{ whiteSpace;
                 reserved "export";
                 ex <- parens (commaSep1 nameOrSymbol );
                 return ex
               }
               <?> "output declarations"

import_decl_vars :: Parser [Data]
import_decl_vars = do{ reserved "import";
                       il <- parens (declCompactList Import);
                       return il
                     }
                   <?> "input declarations"

mkModule :: Identifier->[IdSym]->[Data]->[Statement]->CocoModule
mkModule n exs imps stmts = foldl sortStmt (newModule n exs imps) stmts

newModule n exs imps =
  CocoModule
    { modName=n
    , modExports=exs
    , modStatus=Parsed
    , modOpTable=opTable -- default operation symbol table in Expr module
```

```
    , modOpDefs=[], modDefs= imps,modTypes=Map.empty
    , modACs=[], modImps=[], modErrors=Map.empty
    }

sortStmt :: CocoModule -> Statement -> CocoModule
sortStmt m (StmtDecl ds) = let defs  = modDefs   m  in m {modDefs = ds++defs}
sortStmt m (StmtDef d) = let defs  = modDefs   m  in m {modDefs = d:defs}
sortStmt m (StmtOp d)   = let defs  = modOpDefs m  in m {modOpDefs = d:defs}
sortStmt m (StmtTDef d@(TypeDef id _ _ _) )=
  let defs  = modTypes  m  in
    m {modTypes = (Map.insert id d defs)}
sortStmt m (StmtAC d)   = let defs  = modACs    m  in m {modACs = d:defs}
sortStmt m (StmtImp d)  = let defs  = modImps   m  in m {modImps = d:defs}
```

C.2.2 Statements

Statements are included in the expressions module because all of the higher level modules require the definitions.

Statements in a module are contained within braces, and are terminated by a semi-colon. There are a number of statement types:

typedef A type definition creates a new type.

attributeClass Defines a new attribute class.

operatorDef Defines an interface for a unary or binary operator.

importDef Specifies a list of definitions to import from another module, and defines the mapping between the input and output variables of the module.

declaration The formal declaration of a type constraint for an identifier; there can be multiple type declarations, with the type being the ”meet” of all of the constraints.

definition Defines a complete, partial or functional definition of a data identifier.

There must be at least one statement per module, providing the mandatory output definition.

The data type Statement provides an umbrella type for any possible return type from parsing a statement. The definitions are then split out when the module is created, so the Statement type is not used outside of this module. Because data declaration statements can define multiple identifiers, the results are encapsulated in a list.

Statements within the local body of a definition, in particular function and operator implementations, are limited to variable definitions and declarations.

```
data Statement =
    StmtDecl [Data]
  | StmtDef  Data
  | StmtOp   OperDef
  | StmtTDef TypeDef
  | StmtImp  ImportDef
  | StmtAC   AttrClass
  deriving Show

-- data definition must be last in statement list because it has no distinct label
codeblock = braces (statement `sepEndBy1` semi)

statement :: Parser Statement
statement =  do{ ds<-declaration Private; return $ StmtDecl ds }
         <|> do{d<-typeDef; return $ StmtTDef d}
         <|> do{d<-operatorDef; return $ StmtOp d}
         <|> do{d<-importDef; return $ StmtImp d}
         <|> do{d<-attributeClass; return $ StmtAC d}
         <|> do{d<-definition; return $ StmtDef d}
         <?> "statement"

stmtLocal s =  do{d<-declaration (ExprScope s); return $ StmtDecl d}
           <|> do{d<-definition; return $ StmtDef d}
```

```
        <?> ("declaration or definition in local body of " ++ s)
```

C.2.3 Module Syntax Checking

Beyond the parsing rules, the main syntax requirements of modules is that all data and type definitions are given as ground expressions. Here grounds means one of:

1. a well defined literal value of a recognized type
2. an imported definition with a recognized type
3. expressions of ground expressions based on recognized operator symbols or declared function classes (note that declared is sufficient for functions at this point).
4. a type based on a proper shape, optionally with ground attributes
5. a named type based on a ground type, optionally with ground attributes and dependent parameters, each parameter being of a recognized type
6. a shape is proper if it is one of the predefined HUSC shapes, and any arguments specified for it (e.g. the domain of an array) are of an appropriate type.

Note that the description ground does not imply well-typed, it merely implies a recognition of the identifiers used in the system, and that these definitions are not cyclic (that is part of the checking).

Also note that the shape checking is ”hard-coded”, i.e. is based on fixed type definitions, which allows the syntax checking to terminate.

The checking is done in the following order:

1. check that data definitions (not declarations) are ground - no types required except for literals
2. check that operator definitions (not symbols) refer to recognized symbols and have ground operands

3. check that type definitions are ground - dependent expressions in type descriptions must be ground or parameters from the type defintion.

4. check that attribute class definitions are ground *not implemented*

5. go back and check data declarations are of ground types and ground identifiers

6. check that the type declarations in the operator definitions are ground

This has not been implemented yet and has been added to future work. It is a largely mechanical exercise in traversing the data structures.

Labelling Operator Interfaces

Because the same operator can have multiple interfaces and each has the same name, a unique number is applied to the definitions. This is used when labelling the identifiers. This is true of functions as well, but these are updated when they are inserted into their data tree.

This could probably have been done when the statements were first inserted into the map.

```
chkProgSyntax :: CocoProgram -> CocoProgram
chkProgSyntax prog =
  let mm = mainMod prog in
      CocoProgram (chkSyntax prog mm) (Map.map (chkSyntax prog) (impMods prog))

chkSyntax :: CocoProgram -> CocoModule -> CocoModule
chkSyntax p m = chkTest m

chkTest ::  CocoModule -> CocoModule
chkTest m =
  let m1 = labelDefs m in
    substitute Sub.empty "" m1

labelDefs :: CocoModule -> CocoModule
labelDefs mod =
```

```
  let ndbs = startLabellerST (lblDefs (modDefs mod)) in
    let nops = startLabellerST (lblOps (modOpDefs mod)) in
      mod{modOpDefs = nops,modDefs=ndbs}

lblDefs :: [Data] -> LabellerST [Data]
lblDefs [] = return []
lblDefs ((DataDef id def qs lbody _):bs) =
    do { lbl <- getn id
       ; lbs <- lblDefs bs
       ; return $ (DataDef id def qs lbody lbl) : lbs
       }
lblDefs (b@(DataDecl _ _ _):bs) =
    do { lbs <- lblDefs bs
       ; return $ b : lbs
       }
lblOps [] = return []
lblOps ((OperDef opsym fix def lbody _):bs) =
    do { lbl <- getn opsym
       ; lbs <- lblOps bs
       ; let nlbody = case lbody of
                        (LocalBody dbs)       -> (LabelledBody (opsym++"@"++(show lbl))
                        (LabelledBody id dbs) -> (LabelledBody id  dbs)
       ; return $ (OperDef opsym fix def lbody lbl) : lbs
       }
```

C.2.4 Import Defintions (ImportDef)

A module may load some or all of the exported definitions of another module by importing it. These specify the module to import (by name), a list of definitions that will be imported from the module, and a list of definitions required as inputs by the imported module.

Definitions are imported by their identifier, or if an operator by the symbol. If an operator symbol or function name is specified in the import, then all exported interfaces / implementations associated with that symbol or function name in the

imported module will be imported.

The import defintion may also be qualified by module name, adding the prefix *modname*@ to each identifier. The Boolean and Identifier at the end of the definition are the flag for qualified and module alias respectively.

There is no strategy currently for finding modules within the file system, for now it is assumed that all of the modules needed are supplied in the same file as the main module. That is not intended to be a workable solution, but it is satisfactory for testing purposes.

```
data ImportDef = ImportDef Identifier [IdSym] [Data] Bool Identifier

instance Show ImportDef where
    showsPrec p (ImportDef m exs ims qflag qname) =
           ("\n"++).(("Import module " ++ m ++ " with {") ++).
           (showsList p exs).("} importing {"++).(showsList p ims).('}':).
           (if qflag then (("qualifed as "++qname)++) else id)

instance AllowSubs ImportDef where
    substitute sub scope (ImportDef modname ids defs qflag qname) =
        ImportDef modname (map (substitute sub scope) ids)
                    (substitute sub scope defs)
                    qflag (Sub.apply sub qname)
```

Parsing Module Import Statements

The module import statement is parsed as the name of the module, and export / import definitions.

The concrete syntax for importing modules is :

Use *module* for $(x_1, x_2, \ell\ x_n)$ with
$\{\ y_1 = expr_1$
$;\ y_2 = expr_2$
ℓ
$;\ y_n = expr_n$

}

where x_i is an identifier or operator symbol exported by *module* and *yi* is the identifier of a definition required by *module*.

```
importDef :: Parser ImportDef
importDef =
    do { reserved "Use" <|> reserved "use"
       ; n <- name
       ; reserved "for"
       ; ims <- parens ( commaSep1 nameOrSymbol )
       ; (q,usen) <- option (False, "") (qualModName n)
       ; exs <-option [] (do{ reserved "with"; es <- braces (semiSep definition); return
       ; return $ ImportDef n ims exs q usen
       }

qualModName :: Identifier->Parser (Bool, Identifier)
qualModName mod =
    do { mq <- option False
             (do {(reserved "qualified") <|> (reserved "Qualified"); return True})
       ; qname <- option mod
             (do {reserved "as"; n<-name; return n} )
       ; return (mq, qname)
       }
```

Validating / Managing Imports

Importing a module with a using statement requires that the importer requests only definitions in the export list of the imported module, and provides the definitions required by the imported module. These definitions are selected by identifier, i.e. name or operator symbol. If a function or operator is imported, all interfaces and implementations for that function / operator will be imported (this probably should be controllable with syntax by allowing a list of labels to be given, the labels are already in the concrete syntax for functions/operators).

The import definition is validated during the syntax checking by ensuring the imported definitions are part of the exports, and that the required definitions are supplied to the imported module. These checks are only syntactic, testing for existence. The type system will later ensure the types are compatible.

```
chkImports :: CocoProgram->CocoModule->CocoModule
chkImports (CocoProgram _ modMap) mod =
  let impdefs = (modImps mod) in
    let newErrs = foldl (getImpErr modMap) (modErrors mod) impdefs in
       mod {modErrors = newErrs}

getImpErr :: (ModuleMap)->(ErrorMap)->ImportDef->(ErrorMap)
getImpErr mods errs impd@(ImportDef id _ _ _ _) =
  let impmodM = Map.lookup id mods in
    case impmodM of
      Nothing ->
         let errStr = "Imported module "++id++" not found in file." in
            Map.insert id errStr errs
      Just impmod ->
        let errM = chkImport impmod impd in
          case errM of
            Nothing -> errs
            Just errStr -> Map.insert id errStr errs

-- not implemented, straight forward permutation matching
chkImport :: CocoModule -> ImportDef -> Maybe String
chkImport mod impd@(ImportDef m ims exs _ _) =
  let imperr = "missing from export list: " ++ (show missImps) in
    let experr = "import defintions not supplied: " ++ (show missExps) in
       if (((not.null) missImps) || ((not.null) missExps)) then
           Just $ "module import " ++ imperr ++ "\n" ++ experr
          else Nothing
          where
-- find each identifier in ims in the mod module's export list,
             ims' = foldl (\l x -> case x of {Id n->n:l; Sym _ -> l}) [] ims
```

```
            missImps = filter ((flip notElem) modexs) ims'
-- first strip out symbols and unbox the silly IdSym into Identifiers & Symbols
            modexs = foldl (\l x -> case x of {Id n->n:l; Sym _ -> l}) [] (modExports m
-- pull out each input definition from the import module, make sure there is a def in ex
            modinpdecls = map idFromData (inputDecls (modDefs mod))
            missExps = filter ((flip notElem) (map idFromData exs)) modinpdecls
```

C.2.5 Module Importing and Identifier Lifting

Imported modules supply exported definitions to the importing module, and receive and definitions in their import list. The exported definitions are based on the imported definitions, private definitions and other imported modules.

The current strategy for resolving these imports is to rename all of the identifiers in the module that are not exported with a prefix *modname*@. This raises the definitions to the program context while syntactically hiding them from the other definitions. The exported definitions are not renamed unless the import definition is qualified, and then they are prefixed by the module name and @ figure. These updates are returned in program structure, with the lifted versions of the modules. Note that these are not guaranteed to be unique, and really should be.

This function does not check that the imports and export lists match up, that is done in chkSyntax.

```
mergeMods :: CocoProgram -> CocoProgram
mergeMods (CocoProgram mainmod impmods) =
  let mn = (modName mainmod) in
     CocoProgram (((resImpDef.liftLocal) mainmod){modStatus = Merged})
         (Map.map (resImpDef.liftMod) impmods)

liftLocal:: CocoModule->CocoModule
liftLocal = snd.(rename Sub.empty "")

resImpDef :: CocoModule->CocoModule
resImpDef mod =
```

```
  let imps = modImps mod in
    let sub = foldl Map.union Sub.empty (map impSub imps) in
      let ndefs = concat (map impDefs imps) in
        substitute sub "" (mod {modDefs = (modDefs mod)++ndefs})

impSub :: ImportDef -> Substitution
impSub (ImportDef modname ids defs qflag qname) =
  let idonly = [ n | m@(Id n) <- ids] in
       scopeListSub (liftPrefix modname) idonly

impDefs :: ImportDef -> [Data]
impDefs (ImportDef modname ids defs qflag qname) =
  let sub = scopeListSub (liftPrefix modname) (map idFromData defs) in
    substitute sub modname defs

liftPrefix s = s++"@"

liftMod :: CocoModule->CocoModule
liftMod mod =
  let scope = liftPrefix (modName mod) in
    (snd (rename Sub.empty scope mod)){modStatus = Merged}
```

C.2.6 Displaying Modules

A list of modules can be displayed with dspCocoMods. This will display even if those definitions not in use. At the moment this list is usually the contents of a file, but later might include all of the imported modules.

The display functions will be embedded in a state monad which will contain the module and the current margin setting. The margin setting will move left or right. The monad is defined in miscST.lhs

```
dspCocoProg:: CocoProgram->String->IO ()
dspCocoProg (CocoProgram main others) stage =
   do{ putStrLn stage
```

```
     ; putStrLn "Main module"
     ; dspCocoMod main
     ; mapM dspCocoMod (Map.elems others)
     ; return ()
     }

-- used to pull import declarations (from module header) from map
importFilter (DataDecl _ Import _) = True
importFilter _ = False

dspCocoMod:: CocoModule -> IO ()
dspCocoMod m =
    do { putStrLn $ "module " ++ (modName m) ++ " " ++ (show (modStatus m))
       ; putStrLn $ "exports: " ++ (dspIdSymList (modExports m))
       ; putStrLn $ "type definitions:"
       ; let tds = startDspST (mapM dspTDef (Map.elems (modTypes m)))
       ; mapM putStrLn (concat (map lines tds))
       ; putStrLn $ "data definitions:"
       ; let (idefs,defs) =  List.partition importFilter (modDefs m)
       ; let idbs = startDspST (mapM dspData idefs)
       ; mapM putStrLn (concat (map lines idbs))
       ; let dbs = startDspST (mapM dspData defs)
       ; mapM putStrLn (concat (map lines dbs))
       ; putStrLn $ "operator definitions:"
       ; let ops = startDspST (mapM dspOpDef (modOpDefs m))
       ; mapM putStrLn (concat (map lines ops))
       ; putStrLn $ "importing modules:"
       ; let imps = startDspST (mapM dspImp (modImps m))
       ; mapM putStrLn (concat (map lines imps))
       ; let errcnt = if (Map.size (modErrors m)) > 0
                        then "\nErrors: " ++ (show (Map.size (modErrors m)))
                        else "\nCompilation successful."
       ; putStrLn $ "\nend module " ++ (modName m) ++ " "
                    ++ (show (modStatus m)) ++ errcnt ++ "\n"
```

```
       ; return ()
       }


dspImp :: DspFunc ImportDef
dspImp (ImportDef m exs ims qflag qname) =
  do { let mstr = "Use module " ++ m
     ; let qstr = if qflag then "qualifed as "++qname++" " else ""
     ; let exstr = " for " ++ dspIdSymList exs
     ; tab <- tabST
     ; mrgInST
     ; imstrs <- mapM dspData ims
     ; let with = if (length ims) > 0
                  then "\n" ++ tab ++ " with " ++ (concat imstrs)
                  else ""
     ; tab<-mrgOutST
     ; return $ tab++mstr++qstr++ exstr ++ with
     }
```

C.3 Data Definitions and Declarations

Data is described by data definitions and data type declarations. A data definition associates an identifier, possibly with an associated domain or list of parameters, with an expression. A data type declaration defines a type constraint on an identifier. Data definitions may be complete, partial or functional, but data declarations always constrain the entire identifier (although perhaps this can be relaxed, need to consider this later).

```
module DataDefs where

import Text.ParserCombinators.Parsec
import qualified Data.Map as Map hiding (map)

import MiscST
```

```
import VarSub as Sub
import Lex
import Expr
import Types
import JoinExpr

type DefId = Integer
data Scope = Import | Export | Private | ExprScope String
    deriving Show
data Data =  DataDef  Identifier Def Quantifiers LocalBody DefId
          |  DataDecl Identifier Scope FullType

idFromData  (DataDef id _ _ _ _) = id
idFromData  (DataDecl id _ _)    = id

typeFromDecl (DataDef _ _ _ _ _) = UnknownType
typeFromDecl (DataDecl _ _ ft) = ft

-- used for the types mapping only at this point, may be redundant???
treeSub :: (Substitution->String->a->a) -> Substitution->String-> IdMap a -> IdMap a
treeSub f sub scope =
   let splitSub sub scope = (\x@(k,a)->(Sub.apply sub k, f sub scope a)) in
      ((Map.fromList).(map (splitSub sub scope))).(Map.toList)

inputDecls :: [Data] -> [Data]
inputDecls = filter isInputDecl

isInputDecl (DataDecl _ Import _) = True
isInputDecl                    _  = False

instance AllowSubs Data where
  substitute sub scope (DataDef id def qs lbody lbl) =
     let nid = (Sub.apply sub id) in
         DataDef nid (substitute sub scope def)
```

```
                     (substitute sub scope qs)
                     (substitute sub scope lbody) lbl
  substitute sub scope  (DataDecl id declscope ft) =
     DataDecl (Sub.apply sub id) declscope (substitute sub scope ft)

  rename sub scope db =
    let id = idFromData db in
      let nid = (Sub.apply sub id) in
       let subout = (Map.insert id nid sub) in
         case db of
           (DataDef id def qs lbody lbl) ->
             let newscope = (scope++id++(if (lbl>1) then show lbl else "")++"~") in
              let (lsub1,def2) = rename sub newscope def in
               let (lsub2,newqs) = rename sub newscope qs in
                 let lsub3 = lsub1 `Map.union` lsub2 in
                   let (lsub4,l2body) = rename lsub3 newscope lbody in
                      (subout, DataDef nid (substitute lsub4 newscope def2) qs l2body lb
           (DataDecl id declscope ft) ->
              (subout, DataDecl nid declscope (substitute sub scope ft))

instance Show Data where
    showsPrec p (DataDef v def quants lbody lbl) =
      let lblf = if (lbl > 1) then (showsPrec p lbl) else id in
         (v ++).(showsPrec p def).(showList quants).(showsLocalBody p lbody).(lblf)
    showsPrec p (DataDecl v s t) = (showScope s).(v++).(":"++).(showsPrec p t) where
         showScope Export = ("Export "++)
         showScope Import = ("Import  "++)
         showScope _ = id
```

C.3.1 Data Declarations (DataDecl)

Associates an identifier with a type. The scope identifies when the variable definition is to be exported, provided by the importing module, is private to the module or is

scoped to a local variable.

Variables that are defined as exports must have a definition for the compilation to be successful. Variables that are defined as imports (from the importing module, as opposed to being imported from another module) must not have definitions within the module. These checks are performed by the CheckSyntax module.

Parsing Declarations

A data declaration statement is of the form:

Let *variable* be *type*
Alternately the declaration can be given as a compact declaration *variable* :: *type*. This compact form is used in declaration lists, such as the module header, function interfaces, etc., but can also be given as a general statement.

```
declaration :: Scope -> Parser [Data]
declaration scope =
         do{ reserved "Let"
           ; dList <- declVarList
           ; reserved "be"
           ; t <- fullType
           ; return $ [(DataDecl d scope t) | d <- dList]
           }

declCompactList :: Scope -> Parser [Data]
declCompactList scope
       = do { ds <- commaSep (declCompact scope);
               return $ concat ds
             }

declCompact :: Scope -> Parser [Data]
declCompact scope
       = do{ dList <- declVarList
           ; let unknowns = [DataDecl d scope UnknownType | d <- dList ]
           ; option unknowns ((declTypeVar dList scope) <|> (declTypeExpr dList scope))
```

```
          }

declVarList :: Parser [Identifier]
declVarList = commaSep1 name

declTypeExpr ns scope =
    do { choice[reservedOp "::", reserved "is", reserved "are"]
       ; t <- fullType
       ; return [DataDecl n scope t | n <- ns]
       }

declTypeVar ns scope =
    do { symbol "<:"
       ; t <- fullType
       ; return [DataDecl n scope (VarType n t) | n <- ns]
       }
```

C.3.2 Data Definitions

Data definitions assign an expression to an identifier. There are three kinds of definitions:

Complete completely defines the identifier; there must only be one definition statement for the identifier.

Partial defines the value of the identifier over some subset of it's domain based on quantified variables; there may be multiple partial definitions for an identifier.

Function defines a function interface, and optionally an implementation for that interface.

Definitions are captured in a Def structure, along with the parameter names and types for the function definition, and expressions for the scope of partial definitions.

Complete Definitions

These simply contain the (join) expression represented by the identifier.

Function Definitions

Function definitions consist of an ordered list of parameter declarations (DataDecls) and an expression with an explicit type declaration. The return type will be the same as the type of the expression. The parameters and return type make up a function interface.

A function definition is one interface of a declared function class. The result type and parameter types of the interface must be subtypes of the declared result and parameter types in the class declaration.

The join expression of the definition must have a declared type, and must be defined entirely of variables local to the definition, i.e. parameters and variables defined in the local body of the interface. If no definition is provided for the result expression, for example an undefined variable, then the interface accepted without an implementation. Obviously the program cannot be instantiated without an implementation, but it can be passed through the syntax and type checkers with unsatisfied interfaces.

Partial Definitions

A partial definition can consist of:

Index an index expression applied to a dimensioned type;

Field the name (identifier) of a field in a record.

An indexed partial definition may apply to a single index (Integer) value, or to a range of integer values, defined with quantified variables.

Note that because field labels are isomorphic to a finite natural number domain, we can use the same structure for both concepts. Syntactically, however, a partial definition is currently limited to defining a single field in a record.

The PartialDef structure contains one level of indexing / field dereferencing, then another Def structure which may be the final definition or may be another layer of partial definition.

Only indexing is currently supported, and only to one level (i.e. no arrays of arrays), although this last restriction is easily removed.

There may be multiple partial definitions for any given identifier. The domains must not overlap, and they must also provide a complete definition when combined, i.e. the partial definitions must partition the domain of the identifier. For arrays and iterative definitions, each dimension must be a range of integers with a definition for each value in that range. For records, there must be a definition for each field.

The level of testing for overlapping domains is currently in flux. It is possible with CHRs to do some simple tests, but you really need a full computer algebra system to be complete this way, there may be better strategies.

```
data Def =
      CompleteDef             JoinExpr
    | PartialDef  IdExtension Def
    | FuncDef     [Data]      JoinExpr FullType
    | NullDef

instance AllowSubs Def where
    substitute sub scope (CompleteDef je) =
        CompleteDef (substitute sub scope je)
    substitute sub scope (PartialDef ext def) =
        PartialDef (substitute sub scope ext) (substitute sub scope def)
    substitute sub scope (FuncDef parms je ft) =
        FuncDef (substitute sub scope parms)
           (substitute sub scope je) (substitute sub scope ft)
    substitute _ _ NullDef = NullDef
    rename sub scope (FuncDef parms je ft) =
      let (lsub1,p2s) = liftVars sub scope parms in
        let lsub2 = lsub1 `Map.union` sub in
          let (lsub3, je2) = rename lsub2 scope je in
            (lsub3,FuncDef p2s je2 (substitute (lsub3 `Map.union` lsub2) scope ft))
    rename sub scope def = (sub,substitute sub scope def) -- other defs no parms

instance Show Def where
```

```
    showsPrec p (CompleteDef e) =
        (" = "++).(showsPrec p e)
    showsPrec p (PartialDef idx d) =
        (showsPrec p idx).(showsPrec p d)
    showsPrec p (FuncDef ps e rt) =
        (showList ps).(" = "++).(showsPrec p e).("::"++).(showsPrec p rt)
    showsPrec p (NullDef) = ("null"++)
```

C.3.3 Parsing Definitions

A data definition has a single variable instance on the LHS of an equal sign. Variables are identifiers with an optional set of extensions, either integer indices in square brackets, field labels following a period, or function parameters in parentheses. Indices are parsed as simple integer expressions (a simplified form of expression), field labels as variables, and function parameters as compact declaration lists (i.e. the v::t format). The defFromVar function interprets the LHS variable expression into a definition (Def).

Note that quantifiers can be provided at the beginning of a statement or just before the local body (the ”where” statement).

```
definition:: Parser Data
definition = (try funcdef) <|> vardef
vardef =
    do { q1 <- option [] quantifiers
       ; v <- variable;
       ; reservedOp "="
       ; je <- joinExpr
       ; q2 <- option [] quantifiers
       ; whiteSpace
       ; lbody <- option nullLocalBody (localBody ("coco_" ++ (show (idFromVar v))))
       ; return $ DataDef (idFromVar v) (defFromVar v je) (q1++q2) lbody 0
       }
      <?> "definition"
funcdef =
```

```
    do { f<-identifier
       ; ps <- parens (declCompactList (ExprScope (f++"@")))
       ; reservedOp "="
       ; je <- joinExpr
       ; let rt = getExplicitType je
       ; whiteSpace
       ; lbody <- option nullLocalBody (localBody (f++"@"))
       ; return $ DataDef f (FuncDef ps je rt) [] lbody 0
       }
defIntSimple =
    do { v <- variable;
       ; reservedOp "="
       ; ie <- intExpr
       ; let intft = FullTypeAnnote (GenType "Integer" ScalarShape [])
       ; return $ DataDef (idFromVar v)
                     (defFromVar v (NoJoin (Unconditional (ie, intft)))) [] nullLocalBod
       }
    <?> "definition of variable as an integer expression"

-- only for partial and complete definitions, function Defs should not appear
defFromVar :: Factor -> JoinExpr -> Def
defFromVar (Var id xs) je = pfdef xs je rt
  where
    rt = getExplicitType je
    pfdef [] je _ = CompleteDef je
    pfdef (x@(IdFunc es):xs) je rt = NullDef
    pfdef (x:xs) je rt =
        let p = pfdef xs je rt in
           case p of
              NullDef -> p
              _ -> PartialDef x p
defFromVar _ _ = error "def expects Vars only"

parmFromList::ExprList -> Scope -> Either [Data] String
```

```
parmFromList [] _ = Left []
parmFromList ((Factor (Var n [])):es) scope =
    let eps = parmFromList es scope in
        case eps of
          Left ps -> Left ((DataDecl n scope UnknownType):ps)
          Right s->  Right s
parmFromList (e:es) _ =
    let eps = parmFromList es (ExprScope "") in
        case eps of
          Left ps -> Right $ "Invalid parameter " ++ (show e) ++ ";"
          Right s->  Right $ "Invalid parameter " ++ (show e) ++ "; " ++ s
```

Local Bodies

An expression may have a local body, which is a collection of variable definitions and declarations. These are stored as a list of DefBlocks of class DataDef and DataDecl respectively.

Any variable may have a local body, but for operators and functions this represents and implementation of the interface. The CheckSyntax module ensures that local bodies only contain these two kinds of definitions; new type, attribute class and operator defintions will be ignored.

To rename the variables in a local body, the local identifiers are prefixed with a scope label. The scope is defined by the DefBlock that owns the LocalBody.

```
dataLocal s =  declaration (ExprScope s)
           <|> do{d<-definition; return $ [d]}
           <?> ("declaration or definition in local body of " ++ s)

data LocalBody = LabelledBody {bodyID::Identifier, bodyDefs::[Data]}
               | LocalBody {bodyDefs::[Data]}

nullLocalBody :: LocalBody
nullLocalBody = LocalBody []
```

```
showsLocalBody = showsPrec
instance Show LocalBody where
   showsPrec p (LocalBody []) = id
   showsPrec p (LocalBody lbody) =
     (" where {"++).(showsList p lbody).('}':)
   showsPrec p (LabelledBody _ []) = id
   showsPrec p (LabelledBody bid lbody) =
     (" where " ++).(bid ++).(" {"++).(showsList p lbody).('}':)

localBody :: String -> Parser LocalBody
localBody s =
    do { reserved "where"
       ; label <- option "" name
       ; sts <- braces ((dataLocal s) `sepEndBy1` semi)
       ; return $ if (label=="") then (LocalBody (concat sts))
                      else (LabelledBody label (concat sts))
       }
    <?> "definition of a local body"
```

Variable Renaming in Data Definitions and Declarations

This is where the most complicated aspects of variable renaming are handled. When a set of DefBlocks, such as a local body or a new module, is to be renamed, the first step is to lift the variables (liftVar) defined in the definition block list by prefixing their identifiers with a scope prefix. These new names are added to the substitution, and the new substitution is applied to the definitions in the block list.

Using the Map.union function, which is left biased, allows definitions in the local block to locally eclipse other definitions for the same identifier, i.e. local definitions are of higher priority than higher scope definitions. The new substitution is returned from liftVars, along with the adjusted definition blocks.

C.3.4 Variable Renaming for Scoping

Variables introduced in function definitions, array index definitions or local expression bodies are scoped to that definition. They are lifted to the module level by renaming them, with the scope provided by the renaming. The new name is guaranteed to be unique by prefixing a meaningful label and a " to the name. The label is either the name of the identifier for the definition, or the label provided with the body in the source.

This approach to scoping is not a particularly good one for production use, but is useful for making the trace and intermediate files readable. An alternate strategy should probably be introduced in later phases to deal with identifier lifting.

Local Bodies

A local expression body is of the form

defined = *expression* where *optional id* { *local body* }
Any definitions in the local body are restricted in scope to the local body and the main expression. They are renamed with a scope prefix and then treated as if they were in the module's scope. If the optional body identifier is provided, the prefix is "coco_*id*", otherwise a generated prefix of "coco_*vid*@*nnn*", where vid is the identifier being defined, and nnn is a uniqe identifier to distinguish possible multiple partial definition statements.

functions

In addition to the renaming of local body definitions, the parameters are renamed in the definition and implementation body. The renamed parameters are defined as local variables in the function body. This renaming forces a function value to be dependent only on its formal parameters.

arrays

Arrays are defined through either complete or partial definition statements. In a complete definition, there will be no indexing and no qualifiers, so there are no new

variables. In a partial definition (which may cover the entire domain of the array), there will be between 0 and r new variables introduced, with r being the rank of the array. Each of these variables will be renamed in the LHS and RHS of the definition, but will be scoped to just this definition statement (note: the scope label must therefore take into account that there may be multiple partial definitions for the same identifier label).

substitutions

For each level of scope, there will be a new set of definitions, each of which may have a local body with a new scope. The procedure for renaming is as follows:

- Input is a list of substitutions from the parent definition, and the list of Def-Blocks for the local body. At the module level, the substitution is the identity.
- A new substitution list is prepared from the list of new identifiers defined at this scope level by applying the new prefix to the identifiers (buildSub).
- For each variable declaration, any local body is processed as above, in a depth first substitution, returning the substitutions from that local body. Note that this is only those defined at the next level of scope, the lower levels of substitution are discarded.
- The local body substitution is applied to the variable declaration's expression. This substitution is then discarded (virtually, anyways).
- The current scope substitution is now applied, both to the expression and to the identifier being defined (liftVars). Note that the order of the substitution applications allows local bodies to override higher level scope definitions of the same name; whether this is a good or a bad thing is a matter of taste.

```
liftVarTree :: Substitution->String->IdMap Data-> (Substitution, IdMap Data)
liftVarTree sub scope vmap =
  let vs = Map.elems vmap in
    let (sub2,nvs) = liftVars sub scope vs in
      (sub2, (Map.fromList (map (\d->(idFromData d, d)) nvs)))
```

```
liftVars :: Substitution->String-> [Data] ->(Substitution,[Data])
liftVars topsub scope dbs =
    let cursub = buildSub scope dbs in
       (cursub, snd (rename (cursub `Map.union` topsub) scope dbs) )

buildSub :: String -> [Data] -> Substitution
buildSub scope = (scopeListSub scope).(map idFromData)

scopeListSub :: String -> [Identifier] -> Substitution
scopeListSub scope = (Map.fromList).(map (\s->(s,scope++s)))

instance AllowSubs LocalBody where
    substitute sub scope (LabelledBody id dbs) =
       LabelledBody (Sub.apply sub id) (substitute sub scope dbs)
    substitute sub scope (LocalBody dbs) = LocalBody (substitute sub scope dbs)
    rename sub scope (LabelledBody id dbs) =
       let (localsub, newdbs) = (liftVars sub scope dbs) in
         (localsub, LabelledBody (Sub.apply sub id) newdbs)
    rename sub scope (LocalBody dbs) =
       let (localsub, newdbs) = (liftVars sub scope dbs) in
         (localsub, LocalBody newdbs)
```

Quantifiers

Quantifiers are used in partial definitions to describe the subset of the domain of a dimensioned identifier defined by the current statement. They provide the definition of a variable that is used in the LHS of the partial definition. Currently, the quantified variable must be an integer or an integer range.

There are two kinds of quantifiers:

QuantAll represents a "for all" universal quantifier, where the variable implicitly represents every value possible for the domain of the definition, as defined by its usage in the LHS of the definition.

QuantSome represents a "for $x = expr$ and has a DataDef that explicitly specifies an integer range (sequence) of values the new identifier may take in the definition. In quantifiers of form "$fori = expr$", i should not be a variable in $expr$. This "occurs" check is not currently done, but should be tested in the CheckSyntax module.

When quantifiers are renamed, the identifier is prefixed to scope it to the local definition. This is added to the substitution in a renaming.

```
type Quantifiers = [Quantifier]
data Quantifier = QuantAll Identifier | QuantSome Identifier Data
  deriving Show

instance AllowSubs Quantifier where
    substitute sub scope (QuantAll id) = QuantAll (scope ++ id)
    substitute sub scope (QuantSome id db) =
         QuantSome (scope ++ id) (substitute sub scope db)
    rename sub scope (QuantAll id) =
      let newid = scope ++ id in
        ((Map.singleton id newid), (QuantAll newid) )
    rename sub scope (QuantSome id db) =
      let newid = scope ++ id in
        ((Map.singleton id newid), (QuantSome newid (substitute sub scope db)))


quantifiers :: Parser [Quantifier]
quantifiers =
    do { reserved "For" <|> reserved "for"
       ; qs <- commaSep quantifier
       ; return qs
       }
    <?> "quantifier statement"

quantifier :: Parser Quantifier
quantifier = qualall <|> qualsome
```

```
qualall = do{ reserved "All" <|> reserved "all"
            ; id <- identifier
            ; return $ QuantAll id
            }
qualsome = do{ vd <- defIntSimple
             ; return $ QuantSome (idFromData vd) vd
             }
```

C.3.5 Checking Defintions for Completeness

All variables, operators and function applications appearing in the expressions, guards, or local bodies of a definition must be defined or declared in the module. This state of completeness, where all of the elements of a definition are available, will be called *ground*. This is checked before passing the definitions to the type system and does not guarantee that the definitions are well typed.

An expression is *ground* if:

1. it is a recognized literal value
2. it is a function application, where the function has been **declared** and all of the types are ground.
3. it is a unary or binary operator acting on a ground expression, and the operator symbol is a known symbol
4. it is a defined variable (see below)

A variable is defined to an expression if it is:

defined in the module,

locally defined in the local body of the definition,

imported from another module, either as an input from the importing module / program, or from another module through an import definition, or

quantified in a partial definition statement.

To avoid cycles and duplicate tests, the definition map is encapsulated in a validation map. Checking each identifier's status then proceeds in a depth first manner, marking each identifier being tested with the CurEval flag. Once an identifier is known to be ground, the validation map is updated to reflect this status. If a CurEval flag is encountered, the definitions have a cycle and are flagged as an error.

```
-- ??? syntax checking
```

C.3.6 Displaying Definitions

A definition block describes a HUSC statement. Each type of definition block (DB) has it's own display code.

The module is passed as a parameter so the display can be modified with relevant information. For example, the display of a module that has been parsed but not typed will not include types for the expressions in the graph. Also, any errors will be noted before the description

```
-- display declarations in a comma separated list, usually for parameter lists
dspDataList :: DspFunc [Data]
dspDataList [] = return ""
dspDataList decls =
  do { dbstrs <- mapM dspData decls
     ; let dbstr = foldr1 (\s1 s2 -> s1 ++ ", "++s2) (map (dropWhile ((==) ' ')) dbstrs)
     ; return $ "(" ++ dbstr ++ ")"
     }

dspData :: DspFunc Data
dspData (DataDecl v s t) =
  do{ let scope = case s of {Import->"import -> "; otherwise -> ""}
    ; tstr <- dspFT t
    ; tab <- tabST
    ; return $ tab++scope ++ (dspId v) ++ "::" ++ tstr
    }
dspData (DataDef v def quants lbody lbl) =
  do { let lblstr = case def of
```

```
                      (FuncDef _ _ _)  -> ("<< "++(show lbl) ++ " >> ")
                      (PartialDef _ _) -> ("<< "++(show lbl) ++ " >> ")
                      (CompleteDef _) ->  ""
     ; tab <- tabST
     ; let optnl = if (length quants) > 0 then "\n"++tab else tab
     ; qstr <- (dspQuants quants)
     ; tab2 <- do {case quants of {[]->tabST;(q:qs)-> mrgInST}}
     ; let optnl2 = if (length quants) > 0 then "\n"++tab2 else ""
     ; dstr <- dspDef def
     ; lbstr <- dspLBody lbody
     ; do {case quants of {[]->tabST;(q:qs)-> mrgOutST}}
     ; return $ optnl++lblstr ++ qstr ++ optnl2 ++ (dspId v) ++ dstr ++ lbstr
     }

dspLBody :: DspFunc LocalBody
dspLBody (LocalBody []) = return ""
dspLBody (LabelledBody _ []) = return ""
dspLBody lb =
  do { mrgInST
     ; let (lbl,dbs) = case lb of
                         (LocalBody defs)       -> ("",defs )
                         (LabelledBody id defs) -> (id,defs)
     ; ss <- mapM dspData dbs
     ; tab <- mrgOutST
     ; return $ " where "++lbl++"{\n" ++ (concat ss) ++ "\n" ++ tab ++ "}\n\n"
     }

dspDef :: DspFunc Def
dspDef (CompleteDef je) =
  do { jes <- dspJExpr je
     ; return $ " = " ++ jes
     }
dspDef (PartialDef idx def) =
  do { ds <- dspDef def
```

```
     ; idxstr <- (dspIdExt idx)
     ; return $ idxstr ++ ds
     }
dspDef (FuncDef parmdbs je ft) =
  do { pstr <- dspDataList parmdbs
     ; jes <- dspJExpr je
     ; return $ pstr ++ " = " ++ jes -- don't need rt, it's in the je annotation
     }
dspDef _ = return "=X= (rule or invalid def)" -- no rules yet

dspQuants :: DspFunc [Quantifier]
dspQuants [] = return ""
dspQuants qs =
  do { qstrs <- mapM dspQuant qs
     ; let qstr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) qstrs
     ; return $ "For " ++ qstr
     }
dspQuant (QuantAll id) = return $ "all " ++ (dspId id)
dspQuant (QuantSome id db) = dspData db
```

C.4 Operator Definitions

Operators are functions that can be embedded in expressions. The operator definition statement associates a function interface, including parameter and result types, with an operator symbol. There can be many function interfaces for each symbol, but each symbol may have only one fixity, precedence and associativity setting, so every interface that uses the symbol must share these settings. These fixed settings reflect additional semantic information associated with the operator symbols derived from the HUSC context, namely standard mathematical and scientific notations.

Each operator symbol will, generally, be heavily overloaded. This will range from very different interfaces (e.g. matrix and scalar addition) to identical interfaces with different implementations (adding integer arrays on single, vector and parallel pro-

cessing systems). The evaluation of every expression containing an operator is as the join of all operator interfaces that satisfy the constraints of the expression, meaning that the expression is evaluated to the value of any one interface.

Currently only unary and binary operators are supported, but this could be extended with an appropriate syntax. However, the expression parser in Parsec only allows for unary and binary operators, so the new expression parser (in module Expr) would have to be built from scratch.

Currently, the list of operator symbols is fixed. It should be possible to add new operator symbols, defining the fixity, associativity and precedence. This shouldn't be difficult to add, but it did require that the Parser monad be updated.

```
module OperDef where

import Text.ParserCombinators.Parsec
import qualified Data.Map as Map hiding (map)

import MiscST
import Lex
import VarSub as Sub
import Expr
import Types
import JoinExpr
import DataDefs

data OperDef = OperDef OpSymbol Fixity Def LocalBody DefId

instance AllowSubs OperDef where
  rename sub scope op@(OperDef sym fix def lbody lbl) =
     let newscope =
           case lbody of
             { (LabelledBody id dbs) -> (scope++id++(show lbl)++"~")
             ; (LocalBody dbs) -> (scope++(symname sym)++(show lbl)++"~")
             } in
        let (lsub1,def2) = rename sub newscope def in
```

```
          let (lsub2,l2body) = rename (lsub1 `Map.union` sub) newscope lbody in
              (sub, OperDef sym fix (substitute lsub2 scope def2) l2body lbl)
  substitute sub scope op@(OperDef sym fix def lbody lbl) =
    OperDef sym fix (substitute sub scope def) (substitute sub scope lbody) lbl

instance Show OperDef where
    showsPrec p (OperDef s f fndef lbody lbl) =
           (s++).(showsPrec p lbl).('(':).(showsPrec p f).(')':).
             (showsPrec p fndef).(showsLocalBody p lbody)

symname sym = Map.findWithDefault sym sym opNames
 where
   opNames = Map.fromList
    [ ("+","plus")
    , ("-","neg")
    , ("..","rng")
    , ("*","mult")
    , ("/", "div")
    , ("%", "mod")
    , ("^","exp")
    , ("<", "lt")
    , ("<=", "le")
    , (">=", "ge")
    , (">", "gt")
    , ("==", "eq")
    , ("/=", "ne")
    ]
```

C.4.1 Parsing Operator Definitions

The consist of a symbol and fixity declaration, and a function interface. The parser starts with the operator symbol, then takes parameters to the left and right of the symbol, as required, followed by an equal sign and the result variable. The result

variable is primarily used to specify the return type, and the identifier is locally scoped to the definition. However, if this definition also provides an implementation of the operator, the result variable definition is the value of the result of the operator.

The fixity of the symbol: infix, prefix, postfix; is given by InOp,PreOp and PostOp symbols (named to avoid conflicts with the Haskell infix, prefix and postfix labels). This fixity, along with the precedence and associativity of the operator, must be identical for all interfaces applied to an operator symbol, as defined in the OpSymbolTable for the module.

opdef translates the operator definition into a function definition (provided by Def).

```
-----------------------------------------------------------
-- Operators
-----------------------------------------------------------

operatorDef :: Parser OperDef
operatorDef =
    do { reserved "Operator"
       ; sym <- operator
       ; reserved "as"
       ; loprs <- option [] (parens (declCompactList (ExprScope sym)))
       ; symbol sym
       ; roprs <- option [] (parens (declCompactList (ExprScope sym)))
       ; reservedOp "="
       ; je <- (parens joinExpr) <|> joinExpr
       ; whiteSpace
       ; lbody <- option nullLocalBody (localBody "coco_")
       ; let fix = case (loprs,roprs) of
                   ([],[]) -> NoOp
                   ([],x:xs) -> PreOp
                   (x:xs,[]) -> PostOp
                   (x:xs, y:ys) -> InOp
       ; let opdef = opfdef sym je (loprs++roprs)
       ; return $ OperDef sym fix opdef lbody 0
```

```
       }
    <?> "Invalid operator definition."

opfdef:: Identifier -> JoinExpr -> [Data] -> Def
opfdef _ je ps =
  let rt = getExplicitType je in FuncDef ps je rt

dspOpDef :: DspFunc OperDef
dspOpDef (OperDef s f fndef lbody lbl) =
  do { let opstr = "<< "++(show lbl)++" >> " ++(show f) ++ " " ++ s
     ; argstr <- dspDef fndef
     ; lbodstr <- dspLBody lbody
     ; tab <- tabST
     ; return $ tab++ opstr ++ argstr ++ lbodstr
     }

-- check to make sure the operator symbols and declared types are all valid
-- ??? not implemented
```

C.4.2 Expressions

Expressions may be ntuples, comma separated expressions inside of parentheses. Nested tuples are identified on the basis of parentheses; single expressions on their own or inside of parentheses are not tuples, just simple expressions.

Simple expressions are encoded as an operator with operand(s) or a factor. Operators are either binary with two operand expressions or unary with an associated fixity (PreOp or PostOp). Factors are literals, or variable instances including function applications, array references, and field references (field references not yet implemented, as records were not implemented, but this is easy to do). Factors may also be an expression in parentheses.

```
module Expr where


import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as P
```

```
import Text.ParserCombinators.Parsec.Expr as ParsecExpr
import Text.ParserCombinators.Parsec.Language

import Monad
import qualified Data.Map as Map
import MiscST -- HUSC monads, DspState CounterST and LabellerST
import VarSub as Sub -- HUSC variable renaming class
import Lex

type Identifier = String
type IdMap a = Map.Map Identifier a

type Name = Identifier
data IdSym = Id Identifier | Sym OpSymbol
  deriving Eq
instance Show IdSym where
   showsPrec p (Id x) = (x++)
   showsPrec p (Sym x) = (x++)
instance AllowSubs IdSym where
  substitute sub scope (Id n) = Id $ Sub.apply sub n
  substitute sub scope (Sym s) = Sym s

type OpSymbol = String
data Fixity = PreOp | PostOp | InOp | NoOp
    deriving (Eq, Show)
type OpSymbolTable = ParsecExpr.OperatorTable Char () Expr --  ParsecExpr type

type ExprList = [Expr]
data Expr = Factor Factor | BinOp OpSymbol Expr Expr | UnOp OpSymbol Expr Fixity
          | Tuple [Expr]
    deriving (Show,Eq)
instance AllowSubs Expr where
    substitute sub scope (Factor f) = Factor (substitute sub scope f)
    substitute sub scope (BinOp sym e1 e2) =
```

```
      BinOp sym (substitute sub scope e1) (substitute sub scope e2)
    substitute sub scope (UnOp sym e1 fix) = UnOp sym (substitute sub scope e1) fix
    substitute sub scope (Tuple es) = Tuple (map (substitute sub scope) es)

data Factor = Lit Literal | Var Name [IdExtension] | Parens Expr
    deriving (Show,Eq)
instance AllowSubs Factor where
    substitute sub scope (Lit l) = (Lit l)
    substitute sub scope (Var v exts) =
      Var (Sub.apply sub v) (substitute sub scope exts)
    substitute sub scope (Parens e) = Parens (substitute sub scope e)

idFromVar :: Factor -> Identifier
idFromVar (Var n _) = n
idFromVar _ = error "idFromVar expects Vars only"

variable :: Parser Factor
variable =
    do{ id <- identifier
      ; exts <- many id_extension
      ; whiteSpace
      ; return $ Var id exts
      }
    <?> "Invalid variable definition in factor"

data IdExtension = IdIndex [Expr] | IdFunc  [Expr] | IdField Identifier
    deriving Eq
instance AllowSubs IdExtension where
    substitute sub scope (IdIndex es) = IdIndex (substitute sub scope es)
    substitute sub scope (IdFunc es) =  IdFunc (substitute sub scope es)
    substitute sub scope (IdField id) =  IdField (Sub.apply sub id)
instance Show IdExtension where
  showsPrec p (IdIndex x) = ('[':).(showsList p x).(']':)
  showsPrec p (IdFunc x)  = ('(':).(showsList p x).(')':)
```

```
  showsPrec p (IdField x)  = ('.':).(x++)

id_extension :: Parser IdExtension
id_extension =
     do{ indExs  <- squares intExprList; return $ IdIndex  indExs }
 <|> do{ parmExs <- parens exprList; return  $ IdFunc parmExs }
 <|> do{ reservedOp "."; fieldEx <- identifier; return $ IdField fieldEx }



data Specials = Infinity | NegInf
    deriving (Show,Eq)

data Literal = LitInt Integer | LitReal Double | LitBool Bool
             | LitString String | LitSpecial Specials
    deriving Eq
instance Show Literal where
  showsPrec p (LitInt i) = shows i
  showsPrec p (LitReal f) = shows f
  showsPrec p (LitBool True) = (++) "True"
  showsPrec p (LitBool False) = (++) "False"
  showsPrec p (LitString s) = (++) s
  showsPrec p (LitSpecial Infinity) = (++) "infinity"
  showsPrec p (LitSpecial NegInf) = (++) "-infinity"
```

C.4.3 Parsing Expressions

The operator symbol table is currently hardcoded. In the future, operator symbol definition statements will be added to the operator table dynamically. This will not allows updates to symbols already present.

Integer expressions are a separate class of expressions that are for use in array and iterant variables. Only the integer operators $(+, -, *, \%)$ and the range building operator (..) are included, just what is necessary for integer index expressions or to construct an integer range.

The simple and integer expressions are tested for unary negation operation, and

evaluates it to a negative integer or real value. This is helpful in translating numbers into internal data term representations for the type system.

```
exprList :: Parser [Expr]
exprList = commaSep expr

expr :: Parser Expr
expr = tuplex <|> simpexpr

tuplex :: Parser Expr
tuplex =
  do { es <- parens (commaSep1 expr)
     ; let re = case es of
                 [e] -> e
                 otherwise -> Tuple es
     ; return re
     }

simpexpr :: Parser Expr
simpexpr = buildExpressionParser opTable factor

opTable :: OpSymbolTable
opTable =
    [ [ prefix "-"]
    , [ binop "^" AssocRight ]
    , [ binop "*"  AssocLeft, binop "/"  AssocLeft, binop "%" AssocLeft ]
    , [ binop "+"  AssocLeft, binop "-"  AssocLeft ]
    , [ binop ".." AssocNone]
    , [ binop "==" AssocLeft, binop "/=" AssocLeft, binop "<=" AssocLeft
        , binop "<"  AssocLeft, binop ">=" AssocLeft, binop ">" AssocLeft
      ]
    , [prefix ".not." ]
    , [ binop ".and." AssocLeft]
    , [ binop ".or." AssocLeft ]
    ]
```

```
    where
      binop name assoc
          = Infix   (do{opVar<-try (symbol name); return (\x y -> BinOp opVar x y) }) as
      prefix name
          = Prefix  (do{opVar<-try (symbol name); return (preopeval opVar)})
      postfix name
          = Postfix (do{opVar<-try (symbol name); return (\x -> UnOp opVar x PostOp)})

preopeval "-" =
  \x-> case x of
        (Factor (Lit (LitInt  i))) -> (Factor (Lit (LitInt  (-i))))
        (Factor (Lit (LitReal i))) -> (Factor (Lit (LitReal (-i))))
        _                          -> (UnOp "-" x  PreOp)

intExprList = commaSep1 intExpr

intExpr :: Parser Expr
intExpr = buildExpressionParser intOpTbl intFactor
intOpTbl =
    [ [ prefix "-"]
    , [ binop "^" AssocRight ]
    , [ binop "*"  AssocLeft, binop "/"  AssocLeft, binop "%" AssocLeft ]
    , [ binop "+"  AssocLeft, binop "-"  AssocLeft ]
    , [ binop ".." AssocNone]
    ]
    where
      binop name assoc
          = Infix   (do{opVar<-try (symbol name); return (\x y -> BinOp opVar x y) }) as
      prefix name
          = Prefix  (do{opVar<-try (symbol name); return (preopeval opVar)})

-----------------------------------------------------------
-- Factor
-----------------------------------------------------------
```

```
factor :: Parser Expr
factor =
       do {l <- literal; return $ Factor l}
   <|> do {e <- parens expr; return $ Factor (Parens e)}
   <|> do {v <- variable; return $ Factor v}


intFactor :: Parser Expr
intFactor =
       do {l <- integer; return $ Factor (Lit (LitInt l))}
   <|> do {e <- parens intExpr; return $ Factor (Parens e)}
   <|> do {v <- variable; return $ Factor v}
   <?> "Unknown or invalid factor in integer expression"

-----------------------------------------------------------
-- Literals
-----------------------------------------------------------
literal :: Parser Factor
literal =
        do { n <- naturalOrFloat
           ; case n of
                 Right f -> return $ Lit (LitReal f)
                 Left i  -> return $ Lit (LitInt  i)
           }
    <|> do { reserved "True";  return $ Lit (LitBool True)  }
    <|> do { reserved "False"; return $ Lit (LitBool False) }
    <|> do { reserved "coco_infinity"; return $ Lit (LitSpecial Infinity)}
    <|> do { reserved "coco_neginf"; return $ Lit (LitSpecial NegInf)}
    <|> do { s <- stringLiteral; return $ Lit (LitString s) }
    <?> "unknown literal format"

nameOrSymbol =
     do{ op<-operator; return $ Sym op}
 <|> do{ n<-name; return $ Id n}
```

Expressions are printed as a single line.

```
dspExpr (Factor f) = dspFactor f
dspExpr (UnOp op e PreOp) = do { estr <- dspExpr e; return $ (dspSym op) ++ estr}
dspExpr (UnOp op e PostOp) = do { estr <- dspExpr e; return $ estr ++ (dspSym op)}
dspExpr (BinOp op e1 e2) =
  do {  estr1 <- dspExpr e1
     ;  estr2 <- dspExpr e2
     ; return $ estr1 ++ (dspSym op) ++ estr2
     }
dspExpr (Tuple es) =
  do { estrs <- mapM dspExpr es
     ; return $ "(" ++ (foldr1 (\s t-> s++","++t) estrs) ++ ")"
     }
dspFactor (Lit l) = return (show l)
dspFactor (Var v exts) = do{exstrs <- (mapM dspIdExt exts); return $ (dspId v)++(concat
dspFactor (Parens e) = dspExpr e

dspIdExt (IdIndex xs) =
  do { estrs <- mapM dspExpr xs
     ; let estr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) estrs
     ; return $ "["++estr++"]"
     }
dspIdExt (IdFunc xs) =
  do { estrs <- mapM dspExpr xs
     ; let estr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) estrs
     ; return $ "("++estr++")"
     }
dspIdExt (IdField x) = return $ '.':(dspId x)

-- used for exports lists to merge identifiers and operator symbols
dspIdSymList :: [IdSym]->String
dspIdSymList [] = ""
```

```
dspIdSymList (x:[]) = case x of {Sym s -> s; Id n -> n}
dspIdSymList (x:xs) = (case x of {Sym s -> s; Id n -> n}) ++ (',': (dspIdSymList xs))

--include a display function for identifiers and operator symbols for posterity, just th
dspId = id
dspSym = id
```

C.4.4 Show Instances

Additional instances of Show and some support functions for show instances.

```
showsList p [] = id
showsList p [e] = (showsPrec p e)
showsList p (e:es) = (showsPrec p e).(',':).(showsList p es)
```

A

C.5 Types

Types have three properties in HUSC: a name, shape and optional attributes. The name is primarily a label for a combination of shape and attributes. The shape is mandatory, and one of a fundamental subclass of types defined within HUSC. Shapes may include dependent data or type terms. The attributes are optional values associated with an attribute class defined for the base type. Attribute classes are intended to be extensible, and are treated that way by the parser, so the list of attributes for a type is completely generic, although this has not yet been implemented.

The combination of name, shape and attributes is referred to as a "full type" structure. The FullType structure represents an instance of a type, generally the type of an expression or identifier.

C.5.1 FullType structure

Each class of FullType represents one of:

Generic an identifier, a shape and a list of attributes

Named an identifier, list of dependent arguments and attributes

VarType a type variable with a supertype; no attributes

Type variables have no associated attributes, but instead take on the value of another type during type inferencing, which may contain attributes.

```
module Types where
import Text.ParserCombinators.Parsec
import qualified Data.Map as Map

import MiscST -- display and labelling monads
import VarSub as Sub -- HUSC variable renaming
import Lex
import Expr

data FullType =
    UnknownType
  | GenType     {typeName::Identifier, typeShape::Shape, attrs::AttrList}
  | NameType    {typeName::Identifier, typeDeps::[Expr], attrs::AttrList}
  | VarType     {typeName::Identifier, typeSuper::FullType}


instance AllowSubs FullType where
  substitute sub scope ft =
    case ft of
      UnknownType -> UnknownType
      GenType  n s as -> GenType (Sub.apply sub n) (subs s) (subs as)
      NameType n ds as -> NameType  (Sub.apply sub n) (subs ds) (subs as)
      VarType  n rt -> VarType  (Sub.apply sub n) (subs rt)
   where
      subs x = substitute sub scope x
  rename sub scope UnknownType = (Sub.empty, UnknownType)
  rename sub scope (GenType  n s as) =
    let (lsub1, s2) = rename sub scope s in
```

```
      let (lsub2, as2) = rename (lsub1 `Map.union` sub) scope as in
          ((lsub1 `Map.union` lsub2), GenType (Sub.apply sub n) s2 as2)
  rename sub scope (NameType n ds as) =
    let (lsub1, ds2) = rename sub scope ds in
        let (lsub2, as2) = rename sub scope as in
           ((lsub1 `Map.union` lsub2), NameType  (Sub.apply sub n) ds2 as2)
  rename sub scope (VarType  n rt) =
    let (lsub, rt2) = rename sub scope rt in
       (lsub, VarType  (Sub.apply sub n) rt2)

instance Show FullType where
    showsPrec p (GenType n s attrs) =
      (showsPrec p n).(showsPrec p s).(showAttrList attrs)
    showsPrec p (NameType n dvs attrs) =
      (showsPrec p n).(showsList p dvs).(showAttrList attrs)
    showsPrec p (VarType n rt) =
       ("[typevar]"++).(showsPrec p n).(" <: "++).(showsPrec p rt)
    showsPrec p UnknownType = ("?" ++)
```

C.5.2 Shapes

Shapes are type constructors over the space of scalars. There are five shapes:

Scalar Shape Scalars are individual data items, with subtypes such as numbers, characters and Booleans. It is represented as a unit type, as a scalar type is completely defined by it's name and attributes.

Tuples A tuple is a branching to n full type descriptions ordered by position in the tuple, where n is a fixed value for the tuple. It is represented as a list of full types. The tuple itself may have attributes assigned to it, and each of it's element types may include attributes.

Records are unordered branches represented by set of labelled types. A record type includes a set of field names and their data types. ??? Record types have not yet been implemented ???

DimType Dimensioned types represent a function from a tuple of integers within a domain D subset of Z^n. This is internally defined as a subtype of the generic function shape, and includes subtypes of Arrays, Lists, Sets, and Iterative definitions. The rank and dimensions, stored as a list of integer intervals, are dependent data terms for the array, as is the type of the elements (of course, it is a supertype of the elements).

Function maps a tuple of parameters (full types) to a single return type. The function itself may have attributes, as may each of the types.

```
data Shape =
    ScalarShape
  | RecordShape    [(Identifier,FullType)]
  | TupleShape     [FullType]
  | DimShape       {shDom::[Expr], shapeElem::FullType}
  | FunctionShape  {shParms::[FullType], shapeElem::FullType}

instance AllowSubs Shape where
--      RecordShape flds as -> RecordShape (subs flds) (subs as)
  substitute sub scope sh =
    case sh of
      ScalarShape -> ScalarShape
      TupleShape fts -> TupleShape (subs fts)
      DimShape des rt ->
         DimShape (subs des) (subs rt)
      FunctionShape pts rt ->
         FunctionShape (subs pts) (subs rt)
   where
      subs x = substitute sub scope x

  rename sub scope ScalarShape = (Sub.empty, ScalarShape)
  rename sub scope (TupleShape fts) =
     let (lsub,fts2) = rename sub scope fts in
       (lsub, TupleShape fts2)
```

```
  rename sub scope (DimShape des rt) =
     let (lsub1,des2) = rename sub scope des in
       let (lsub2,rt2) = rename (lsub1 `Map.union` sub) scope rt in
         (lsub1 `Map.union` lsub2, DimShape des2 rt2)
  rename sub scope (FunctionShape pts rt) =
     let (lsub1,pts2) = rename sub scope pts in
       let (lsub2,rt2) = rename (lsub1 `Map.union` sub) scope rt in
         (lsub1 `Map.union` lsub2, FunctionShape pts2 rt2)

instance Show Shape where
{-    showsPrec p (RecordShape dbs attrs) =
      ("Record"++).(showAttrList attrs).(showList dbs) -}
    showsPrec p (TupleShape ts) =
      ("[tuple]"++).(showList ts)
    showsPrec p (ScalarShape) = id
    showsPrec p (DimShape des et) =
      ("[dim]"++).(showsList p des).(" of "++).(showsPrec p et)
    showsPrec p (FunctionShape ps rt) =
      (showList ps).("->"++).(showsPrec p rt)
```

Attributes

An attribute is an instance of an attribute class (the Identifier), with the attribute value stored as an expression of the property type of that class.

Variable renaming just requires a substitution be applied to the attribute class and expression.

```
type AttrList = [Attribute]
data Attribute = Attribute Identifier Expr
  deriving Show
instance AllowSubs Attribute where
    substitute sub scope (Attribute ac val) =
         Attribute (Sub.apply sub ac) (substitute sub scope val)
    rename sub scope (Attribute ac val) =
```

```
      let (lsub, val2) = rename sub scope val in
        (lsub, Attribute ac val2)

-- customizing show for attribute lists to get braces instead of squares
showAttrList [] = id
showAttrList attrs = ('{':).(showsList 0 attrs).('}':)
```

C.5.3 Parsing Type Expressions

Parsing of types and shapes are interwoven in the syntax.

```
fullType :: Parser FullType
fullType =
      multiType
  <|> singleType

multiType = (try recordType) <|> tupleType

recordType =
  do { flds <- parens( commaSep fldDef )
     ; as   <- option [] attributeList
     ; return $ GenType "" (RecordShape flds) as
     }
fldDef =
  do { n <- name
     ; reservedOp "::"
     ; ft <- fullType
     ; return (n,ft)
     }
tupleType =
  do { flds <- parens( commaSep fullType )
     ; as   <- option [] attributeList
     ; return $ GenType "" (TupleShape flds) as
```

```
    }

singleType =
  do { n  <- identifier
     ; dps <- option [] exprList
     ; as <- option [] attributeList
     ; case dps of
         []     -> (   try (typeVar n)
                   <|> do{s<-shape; return $ GenType n s as} )
         (x:xs) -> return $ NameType n dps as
     }
typeVar n =
  do { whiteSpace
     ; reservedOp "<:"
     ; t <- fullType
     ; return $ VarType n t
     }

shape = dimType <|> (try funcType) <|> return ScalarShape
dimType :: Parser Shape
dimType =
  do { ds <- squares intExprList
     ; reserved "of"
     ; rt <- fullType
     ; return $ DimShape ds rt
     }
funcType :: Parser Shape
funcType =
  do { whiteSpace
     ; reserved "from"
     ; ts <- parens (commaSep fullType)
     ; reserved "to"
     ; rt <- fullType
     ; return $ FunctionShape ts rt
```

```
    }

attributeList:: Parser [Attribute]
attributeList = option [] (braces (commaSep attribute))

attribute:: Parser Attribute
attribute =
    do{ ac  <- name
      ; val <- option (Factor (Lit (LitBool True))) (parens expr)
      ; return $ Attribute ac val
      }
```

C.5.4 Printing Types

Printing of named types and type variables is straightforward. For generic types, the name of the type is printed first, then any shape specific information, then the attribute list. This works conveniently for tuples / records because they have name.

```
dspFT :: FullType -> DspST String
dspFT (GenType n s as) =
  do { shstr <- dspSh s
     ; astr <- dspAttrs as
     ; return $ (dspId n)++astr++shstr
     }
dspFT (NameType n dps as) =
  do { dpstrs <- mapM dspExpr dps
     ; let dstr = "("++(foldr1 (\s1 s2 -> s1 ++ (',':s2)) dpstrs)++")"
     ; astr <- dspAttrs as
     ; return $ (dspId n)++(if (length dps) > 0 then dstr else "")++astr
     }
dspFT (VarType n spt) =
  do { sptstr <- dspFT spt
     ; return $ (dspId n)++"<:"++sptstr
     }
```

```
dspFT UnknownType = return "?"

dspSh :: Shape -> DspST String
dspSh (RecordShape flds) =
  do { return $ "Records not implemented"}

dspSh (TupleShape ts) =
  do { fldstrs <- mapM dspFT ts
     ; let fldstr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) fldstrs
     ; return $ "("++fldstr++")"
     }
dspSh ScalarShape = return $ ""
dspSh (DimShape ds et) =
  do { dstrs <- mapM dspExpr ds
     ; let dstr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) dstrs
     ; etstr <- dspFT et
     ; return $ "["++ dstr ++ "]" ++ " of "++ etstr
     }
dspSh (FunctionShape ps rt) =
  do { pstrs <- mapM dspFT ps
     ; let pstr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) pstrs
     ; retstr <- dspFT rt
     ; return $ "("++ pstr ++ ") ->" ++ retstr
     }


dspAttrs :: [Attribute]-> DspST String
dspAttrs [] = return ""
dspAttrs as =
  do { astrs <- mapM dspAttr as
     ; let astr = foldr1 (\s1 s2 -> s1 ++ (',':s2)) astrs
     ; return $ "{" ++ astr ++ "}"
     }
dspAttr (Attribute ac e) =
```

```
  do { estr <- dspExpr e
     ; return $ ac ++ "(" ++ estr ++ ")"
     }
```

C.6 Type Definition

A new type is assigned an identifier, optionally with a list of dependent type expressions (as DataDefs), the type (FullType) it is based on, and a Boolean flag indicating if the new type is a subtype (True). If it is not a subtype, it is an incompatible type with the same basic structure (False).

The new type may be identical to an existing or generic type, or it may refine the base type in some way. Type identifiers are by convention capitalized, but this is not necessary. Types must be exported from a module explicitly if they are to be made available to importing modules.

```
module TypeDef where

import Text.ParserCombinators.Parsec
import qualified Data.Map as Map hiding (map)

import MiscST
import Lex
import VarSub as Sub
import ValidMap
import Expr
import Types
import DataDefs

data TypeDef = TypeDef Identifier [Data] Bool FullType

instance AllowSubs TypeDef where
  rename sub scope td@(TypeDef id dbs st ft) =
    let nid = Sub.apply sub id in
      let sub1 = Map.insert id nid sub in
```

```
        let (depsub,ndbs) = rename sub1 scope dbs in
          (sub1,TypeDef nid ndbs st (substitute (depsub `Map.union` sub1) scope ft))
  substitute sub scope td@(TypeDef id dbs st ft) =
    let nid = Sub.apply sub id in
      let sub1 = Map.insert id nid sub in
        let (depsub,ndbs) = rename sub1 scope dbs in
          TypeDef nid ndbs st (substitute (depsub `Map.union` sub1) scope ft)

idFromType = (\d@(TypeDef id _ _ _) -> id)

liftTypeTree :: Substitution->String->IdMap TypeDef-> (Substitution, IdMap TypeDef)
liftTypeTree sub scope tmap =
  let ts = Map.elems tmap in
    let (sub2,nts) = liftTypes sub scope ts in
      (sub2, (Map.fromList (map (\d->(idFromType d, d)) nts)))

liftTypes :: Substitution->String-> [TypeDef] ->(Substitution,[TypeDef])
liftTypes topsub scope tds =
    let cursub = buildTSub scope tds in
       (cursub, substitute (cursub `Map.union` topsub) scope tds )

buildTSub :: String -> [TypeDef] -> Substitution
buildTSub scope = (scopeListSub scope).(map idFromType)

--scopeListSub from datadefs, creates substitution by adding scope to ids

instance Show TypeDef where
   showsPrec p (TypeDef t ps st bt) =
      let stf = if (not st) then ("based on "++) else id in
           ("Type "++).(showsPrec p t).(showsList p ps).(" as "++).(stf).(showsPrec p bt
```

Type definitions are of the form

Define *typename* [*args*]* as [*basedon*]? *basetype*
The new name is the label for the type. If there are arguments provided, these are dependent type variables used in the basetype definition, for instance

Define Matrix(m,n,a) as Array[1..m,1..n] of a
These variables are only in scope within the type definition.

A new type by default is assumed to be a subtype of the original type. Alternately, including the key words "based on" before the base type will make the new type entirely distinct and incompatible with the base type.

```
typeDef :: Parser TypeDef
typeDef =
  do { reserved "Define"
     ; t <- name
     ; ps <- option [] (parens (declCompactList (ExprScope ("TypeDef " ++ t))))
     ; reserved "as"
     ; st <- option True (do {reserved "based" <|> reserved "Based"; reserved "on"; retu
     ; bt <- fullType
     ; return $ TypeDef t ps st bt
     }

dspTDef (TypeDef t ps st bt) =
  do { pl <- dspDataList ps
     ; let s1 = "Type " ++ (dspId t) ++pl
     ; let sep = if (length s1) > 30 then "\n    " else " "
     ; let s2 = if (not st) then ("variant of ") else "<: "
     ; tab <- tabST
     ; btstr <- dspFT bt
     ; return $ tab++s1 ++ sep ++ s2 ++ btstr
     }
```

C.6.1 Valid Type Definitions

A type definition is valid if it can be resolved, with the appropriate application of well defined dependent expression values, into a generic type. A generic type is one of the predefined shapes (Scalar, DimType, Function, Tuple, Record) optionally with attributes of defined classes. A type is resolved to a genric type if tracing the subtype / based on hierarchy of the type definitions results in a generic type. The dependent arguments for a type are well defined if each of them is a well defined type. Note that a type with dependent arguments cannot have an argument of it's own type, nor can the types form a cycle.

If an error is found in the type definition checks, it is recorded in the error map of the CocoModule structure.

Validation is carried out across an adapted tree called a validation tree. The validation tree is the type map with a three value condition flag. The conditions are:

Unknown the type has not been considered for validity.

Valid the type has been shown valid.

Invalid the type definition has been shown to be flawed, the error is attached.

CurEval the type is currently being evaluated for validity; if this turns up it means a cycle has been established in the validity test.

The tests are folded over the validity map, collecting results and errors. These errors are finally removed from the validity map and returned. It is assumed that the calling program will interpret the error map and either exclude invalid types from further processing or interrupt the process.

```
type TypeMap = IdMap TypeDef
emptyTypeMap = Map.empty

chkTypes:: TypeMap -> ErrorMap Identifier
chkTypes tm =
  let tvalm = mkValMap tm chkType in
    let ts = Map.keys tm in
```

```
      let tfinalm = foldl valUpdate tvalm ts in
         errsFromValMap tfinalm

chkType :: ValMapDTest Identifier TypeDef
chkType vmap@(VMap vm tm dtest) t =
  let mtd = Map.lookup t tm in
    case mtd of
       Nothing -> NotFound t
       Just td@(TypeDef tn dds st ft) -> chkFT vmap ft

chkFT:: (ValMap Identifier TypeDef)-> FullType -> (ValMapStatus Identifier)
chkFT vmap@(VMap vm tm dtest) ft =
   case ft of
      (GenType id _ as)      -> chkAttrs vmap as
      (VarType id sft)       -> chkFT vmap sft
      (NameType id deps as) ->
         let sts = valTest vmap id in
           case sts of
             Valid ->
               let mtd = Map.lookup id tm in
                 case mtd of
                   Nothing -> NotFound id
                   Just td@(TypeDef _ dps _ _) ->
                     if ((length dps)==(length deps)) then Valid
                        else Invalid ("type "++id++" declares " ++
                                       (show (length dps)) ++ " dependent arguments "++
                                       " but is defined with "++(show (length deps)))
             Unknown -> Invalid ("unable to resolve defintion for type "++id)
             NotFound n -> Invalid ("type "++n++" has no definition.")
             CurEval -> Invalid ("type "++id++" is cyclically defined.")
             Invalid err -> Invalid err

-- have not implemented this as attribute classes are hard coded right now
chkAttrs _ [] = Valid
```

```
chkAttrs vmap ((Attribute ac prop) : as) = Valid
```

C.7 Attribute Classes

Attribute classes represent semantic information associated with terms that supplements the basic shape information of the term. The attribute class definition provides the following information:

Class is an identifier providing the class name.

persistant is a Boolean value indicating whether an attribute class, once introduced in a module, must be explicitly addressed by a rule in every instance it appears for compilation to be successful (default is False).

Target is the type to which the attribute may be applied; the attribute also applies to all subtypes.

Property is the type of the value associated with the attribute.

Body is a collection of rules, predicates and local variable definitions / declarations that provide semantics for the attribute class.

Note that these attribute classes are not currently being used, the translation to the type system is done manually for the supported attributes. As a result, the structure provided here has not been validated. The entire definition of attribute classes has been promoted to a future research item.

```
module AttrClass where

import Text.ParserCombinators.Parsec
import qualified Data.Map as Map hiding (map)

import MiscST
import Lex
import VarSub as Sub
import Expr
```

```
import Types
import DataDefs

data AttrClass=
    AttrClass {attrClass::Identifier, persistant::Bool,
               target::FullType, property::FullType, body::AttrClassBody}

instance AllowSubs AttrClass where
  substitute sub scope ac =
    ac { attrClass = (Sub.apply sub (attrClass ac)),
         target = (substitute sub scope (target ac)),
         property = (substitute sub scope (property ac))
-- must also do body, but I need to think about this ???
       }

instance Show AttrClass where
    showsPrec p (AttrClass n pers trg prop body) =
                ("\n"++).("AttrClass "++).(n++).('(':).(showsPrec p prop).(')':).
                (" of "++).(showsPrec p trg).(':':).(showsPrec p body)


attributeClass :: Parser AttrClass
attributeClass =
    do { reserved "Attribute"
       ; (reserved "Class" <|> reserved "class")
       ; pers <- option False
                    (do {(reserved "Persistant" <|> reserved "persistant"); return True})
       ; n <- identifier
       ; propType <- option (GenType "Boolean" ScalarShape []) (parens fullType)
       ; whiteSpace
       ; reserved "of"
       ; targType <- fullType
       ; sts  <- braces (dataLocal ("AttrClass " ++ n))
       ; return $ AttrClass n pers targType propType (attrBody n sts)
```

```
}
```

C.7.1 Attribute Class Body (AttrClassBody)

The body of the attribute class contains the semantic information about the attribute in terms of rules and function definitions. Some of these are mandatory, some are optional, and some allow more than one contribution.

sub*Class* is a predicate (Boolean function) over two property values of the attribute, evaluating to True if the first property value represents a subattribute of the second value.

is*Class* Optional (zero or more) predicate functions over the object the attribute is attached to (target) and the attribute property value that evaluate to true if the attribute holds at that value. These predicates are made available for run-time confirmation that an attribute holds. If no such functions are provided, there can be no run time testing for an attribute. If more than one is provided, they will be used in a join relationship, i.e. the value of the function will be the value of any one of the provided predicates.

Class Propagation (derivation and inference) rules that define when an attribute or attribute constraint is generated based on operators or functions, independently of the interface or implementation. The rule begins with function called *ClassName* using pattern matching from one expression to another expression.

```
data AttrClassBody = AttrClassBody {subAttrTest::Def, isAttrTest::Def,
                                    attrRules::[Def], locals::[Data]}
instance Show AttrClassBody where
    showsPrec p (AttrClassBody sub is rules locals) =
        ("\n{"++).("subAttr: "++).(showsPrec p sub).
                   (";\n isAttr: " ++).(showsPrec p is).
                   (";\n rules :"++).(showList rules).
                   (";\n locals :"++).(showList locals).("}\n"++)

attrBody :: Identifier -> [Data] -> AttrClassBody
```

```
attrBody attrcl ds = foldl (afs attrcl) nullAttrBody ds where
    nullAttrBody = AttrClassBody NullDef NullDef [] []
    afs ac body (DataDef n d qs l _)
      | n == ("is"++ac)   = body {isAttrTest = d}
      | n == ("sub"++ac)  =
              case  (subAttrTest body) of
                NullDef -> body {subAttrTest = d}
                _ -> body -- ignore extra subAttr tests when supplied, should really iss
      | n == ac           = body {attrRules = d : (attrRules body)}
    afs _  body  d        = body {locals = d:(locals body)}
```

C.7.2 Displaying Attribute Classes

```
dspAC :: DspFunc AttrClass
dspAC (AttrClass n pers trg prop body) = return ("AttrClass "++n)
```

C.8 Type System Encoding

The type system is implemented in SWI Prolog and Constraint Handling Rules (CHRs). This module handles the transition from the Haskell based structures to Prolog relations and CHR constraints.

There are four fundamental data objects in the type system: types, variables (or identifiers), expressions and attributes. Type definitions are provided as individual clauses. The variable definitions and type declarations are provided as a goal, consisting of constraints (a special kind of predicate). The objective of the type system is to provide a minimal supertype for each of the variables and expressions, this being the least general type description that allows for all possible values the variable might take. The type description includes both a shape, possibly with dependent terms associated, and a set of attributes.

The goal of the system should always result in success, meaning types are assigned to expressions and identifiers, but the types themselves may express errors. In particular, the type **TypeBot** (bottom) indicates that this expression or definition is only valid over a null solution space, i.e. is untypable. Alternately, abstract types

such as **TypeTop** or underspecified types such as the generic function with arbitrary parameters, indicate the definition is underspecified and the result could be anything. Interpretation of these results occurs in the type system decoding module.

It is assumed that before the encoding stage, all identifiers have been renamed to allow them to be lifted to the module level without name conflicts. This is carried out in the CocoSyntax module.

Expressions and Labels

Every data term, literal, factor, expression, etc. is encoded in a separate predicate. These are labelled with a unique arbitrary expression label, and each expression will be assigned it's own type constraints and type values. The type system will resolve the best choice of type for each expression, generally by first propagating the type constraints and then deriving the types of compound expressions from the operand expressions, although the rule system should actually be confluent (this has not been proven or tested).

Expression labels are generated using the CounterST monad, which provides a uniquely numbered label, given a prefix. The prefix may be changed, resetting the count to 0. Generally, expression labels include the name of the identifier they are associated with and are numbered in depth first pass. There is no dependency on the label, other than it is unique.

```
module EncTypes where

import qualified Data.Map as Map hiding (map)
import Monad

import MiscST
import Lex
import VarSub as Sub
import Expr
import Types
import JoinExpr
import DataDefs
```

```
import OperDef
import TypeDef
import AttrClass
import HUSCModule

-- encapsulate identifiers in single quotes to allow uppercase, special symbols
atomid s = "'"++s++"'"
-- semi-encapsulate a prefix for atoms
atompre s = ('\''):(s++"@")
-- encapsulate identifier as Prolog variable for inclusion in type/4 relations.
provarid id = '_':id
atomCombine s1 s2 = ns1 ++ ns2 where
    ns1 = reverse $ dropWhile ('\''==) $ reverse s1
    ns2 = dropWhile ('\''==) s2
```

C.8.1 Encoding Modules

Modules are encoded as a list of strings, each containing an individual Prolog / CHR predicate. This includes a predefined set of types that are core to the system, namely the generic shapes [1]. These are hardcoded in the list below (HUSCpreloads) A description of the preloaded types is available as comments in the appropriate prelude files.

```
huscPreloads =
  [ "type(t('Top',[],[]),t('Top',[],[]),[],bt)"
  , "type(t('Tuple',[dt(int,N), tt(cocontpl(N,t('Top',[],[])))],[]),t('Top',[],[]),[],st
  , "type(t('Function',[tt(_),dt(int,Ps),tt(cocontpl(Ps,t('Top',[],[])))],[]),t('Top',[]
  , "type(t('DimType',[tt(_), dt(int,R), dt(cocontpl(R,interval(int)),_)],[]), t('Top',[
  , "type(t('Scalar',[],[]),t('Top',[],[]),[],st)"
  ]

encMod :: CocoModule -> [String]
encMod mod =
```

[1]The only type that is actually necessary is Scalar, and only if DimTypes are to be used, but the language is intended to use all of them

```
huscPreloads
  ++ concat (map encTypeDef (Map.elems (modTypes mod)))
  ++ concat (map encData  (modDefs mod))
  ++ concat (map encOpDef (modOpDefs mod))
```

C.8.2 Definition Encoding Overview

Definition blocks are generic data structures that store the information from the lines of the code modules. The following block types are found:

TypeDef defines a new type by name, including dependent variables and new attributes, relative to an existing type.

DataDecl declares the type of a data identifier, which could be any type including a function.

DataDef provides a definition for an identifier; it could be a partial definition or the definition of a function implementation, and there may be multiple variable definitions for identifiers where appropriate.

OperDef defines an interface for an operator symbol, giving the operand and result types; there will be many definitions for each given symbol.

ImportDef imports definitions from another (not the importing) module; nothing has to be done to encode them because the module merging and renaming has renamed all of the variables to match at the program scope.

AttrClass describes the properties and rules for a new attribute class; **this has not yet been implemented, the attributes are hard coded**.

C.8.3 New Type Definitions

Types are stored in a predicate

$\mathrm{t}(name,\ [depvars]^*)$ or $\mathrm{t}(name,\ [depvars]^*,\ [attrs]^*)$

where [*depvars*] is a comma separated list of dependent terms for the type and *attrs* is a comma separated list of attributes. This predicate must either be in a type declaration predicate (below) or refer by name to a defined type, with an appropriate dependent variable representation.

The relationship between types are predicates

type(*type*, *basetype*, [*attributes*]*, [st | bt])

where

- *type* is a t(name,depvar) fact naming the new type and it's dependent terms
- *basetype* is the t(name,depvar,attrs) relation of the base type
- *attributes* is a list of additional attributes that apply only to the new type, in addition to those already defined for the base type.
- st / bt flag, which indicates whether this type is a subtype or a new type based on the old type, but not to be treated as a subtype of it

C.8.4 Dependent Variable Encoding

The dependent variables are either type variables or data terms. Type variables are identifiers followed by ̈¡:̈then a type, indicating they are subtypes of that type. Data variables are regular variable declarations of the form *var* :: *type*. Data variables are either expressible in the internal data term types, those that can be manipulated within the type system, or not expressible internally, so are recorded as pure HUSC types. Dependent type and data terms are encoded in tt (type term), dt (data term) and gt (general term) predicates.

The internal data terms can be of the following types and values:

bool, 0 or 1

int, acceptable prolog integer

float, acceptable prolog float

interval(A), (v1,v2) where v1 and v2 are of internal type A

ntuple(n,A), [v1,...,vn] where v1,...,vn are of (identical) internal type A.

list(A),[v1,...] where v1,... are of internal type A, for an arbitrary length list.

Data terms are manipulated within the type system, including evaluating terms through equality and less / greater than relations. In addition, HUSC variables and literals in a representable type are also mapped to the internal data term system. Where possible these are translated into the internal data term language for the type system. Where this is not possible, they are encoded as expressions and dumped in the list, but these can't currently be used for anything.

This approach should be modified by replacing the internal data term system with a full HUSC interpreter and CHR subsystems to accomplish a number of additional tasks within that. That will require a certain level of bootstrapping, and will have to wait for future versions.

Dependent Variables in Type Definitions

Dependent variables encoded in type expressions are recorded as Prolog variables, allowing them to be matched to their base types by Prolog. Accordingly, all of the identifiers are preceded by an underscore (_) to guarantee that they are syntactically understood to be Prolog variables.

Dependent variables and expressions in other locations are recorded as atoms, and are manipulated by the internal data term system exclusively.

```
depVarsFromData :: [Data] -> CounterST (String,[String])
-- depVarsFromData _ = return ("",[])

depVarsFromData [] = return ("",[])
depVarsFromData (db:dbs) =
    do { (s,l) <- depVarFromData db
       ; (ss,ls) <- depVarsFromData dbs
       ; return (s++ss, l++ls)
       }
```

```
depVarFromData (DataDecl id scope ft) =
     case ft of
         (GenType st _ _) -> return ("dt("++(st2it st)++","++(provarid id)++")",[])
         (NameType st _ _) -> return ("dt("++(st2it st)++","++(provarid id)++")",[])
         (VarType n rt) ->
             do { (rts,trels) <- encFT rt
                ; return ("tt(tv("++n++","++rts++"))",trels)
                }

depVarFromData db =
  return ("error(Unexpected definition in dependent variable list for "
              ++ (idFromData db)++")",[])
```

Dependent Expressions in Types

A dependent expression is an instance of a

```
depExprList :: [Expr] -> [String]
depExprList = map depExpr

depExpr :: Expr->String
depExpr (Factor (Lit l)) =
  case l of
     (LitInt i)  -> "dt(int,"++(show i)++")"
     (LitReal f) -> "dt(float,"++(show f)++")"
     (LitBool True) -> "dt(bool,1)"
     (LitBool False) -> "dt(bool,1)"
     (LitString s) -> "dt(string,"++s++")"
     (LitSpecial Infinity) -> "dt(int,coco_inf)"
     (LitSpecial NegInf)  -> "dt(int,coco_ninf)"
depExpr (Factor (Var x [])) = "dt(int,"++x++")" -- ??? temporary, all vars are integers
depExpr (BinOp ".." (Factor (Lit (LitInt i1)))
          (Factor (Lit (LitInt i2))) ) = "dt(interval(int),("++(show i1)++","++(show i2)
depExpr (BinOp ".." (Factor (Lit (LitReal f1)))
```

```
         (Factor (Lit (LitReal f2))) ) = "dt(interval(int),("++(show f1)++","++(show f2
depExpr _ = "dt(null,[])"  -- ??? temporary, everything else is null

st2it st =
   case st of
       "Integer" -> "int"
       "Real" -> "float"
       "Boolean" -> "bool"
       "String" -> "string"
       "Interval(Integer)" -> "interval(int)"
       "Interval(Real)" -> "interval(float)"
       otherwise -> "error"
```

Variable Declarations

An identifier is declared explicitly to be of a type using **vdecl**(*id*,*type*). This declared type will be a constraint on the definition of the identifier, guaranteeing that it will be a subtype of the declared type. It does not specify that the identifier is exactly that type, as there may be additional information generated by the use of the identifier.

An extra list appears as the third parameter in the type relation. This was originally intended for attributes, but they are in the base type instead. This extra list should be removed at some point.

```
encTypeDef :: TypeDef -> [String]
encTypeDef (TypeDef id dbs st ft) =
    startCounterST id
      ( do { (dtlbl,dtrels) <- depVarsFromData dbs
           ; (btlbl,btrels) <- encFT ft
           ; let td = "t("++(atomid id)++",["++dtlbl++"],[])"
           ; let sbts = if st then "st" else "bt"
           ; return $ ("type("++td++","++btlbl++",[],"++sbts++")"): (dtrels ++ btrels)
           } )

encData :: Data -> [String]
```

```
encData (DataDecl id scope ft@(GenType _ (FunctionShape pts rt) _)) =
  let lbl = id ++ "_dcl"  in
    startCounterST lbl
      ( do { elbl <- getnext
           ; (rtlbl,rtrels) <- encFT rt
           ; (ptlbls,ptrels) <- encFTList pts
           ; let ptslbl = wrap ptlbls
           ; let fc = "funclass(t('"++id++"',["++rtlbl++",dt(int,"++(show (length pts))+
           ; let fsh = "t('Function',["++rtlbl++",dt(int,"++(show (length pts))++"),["++
           ; return $ (fc++","++fsh) : rtrels++ptrels
           } ) where
                 wrap = foldr (\e s-> ",tt(" ++ e ++ ")" ++ s) ""

encData (DataDecl id scope ft) =
  let lbl = id ++ "_dcl"  in
    startCounterST lbl
      ( do { elbl <- getnext
           ; (tlbl,trels) <- encFT ft
           ; let vd = "vdecl('"++id++"',"++tlbl++")"
           ; case scope of
               Import ->
                 let vid = "vdef('" ++id++"',cocoinput('"++id++"'))" in
                   return $ vd : (vid : trels)
               otherwise -> return $ vd : trels
           } )
```

Variable, Function and Operator Definitions

Variable definitions are recorded through the predicate **vdef**(*id*,*exprl*), where *exprl* is the generated unique label for the expression. This is followed by the encoded expression for the definition, the encoded quantifier variables if any, and the encoded local body. Local bodies are blocks of identifier definitions and type declarations (no type, operator nor attribute class definitions allowed), which have been renamed to

raise them to the module scope, so this can be handled through simple recursion.

Quantifiers are used in partial definition statements to describe the subset of the domain of the identifier covered by a definition. Partial definitions may be used for arrays, lists and iterative types. The identifier given (e.g. i in for all i) is treated as a local variable for the definition, part of the local body of the expression.

Operator definitions are just a syntactic variant of a function definition. The operator symbol is encapsulated in single quotes so it can be a valid Prolog atom.

```
encData (DataDef id def qs lbody cntr) =
  let lbl = atomCombine id (if (cntr > 1) then (('@'):(show cntr)) else "")  in
    startCounterST lbl
      ( do { elbl <- getnext
           ; (drt,drels) <- encDef id elbl def
           ; qrels <- encQuants qs
           ; let lbrels = encLBody lbody
           ; return $ ("vdef('"++id++"',"++elbl++")"): (drels ++ qrels ++ lbrels)
           } )

encLBody :: LocalBody -> [String]
encLBody (LocalBody dbs) = concat (map encData dbs)
encLBody (LabelledBody _ dbs) = concat (map encData dbs)
```

Quantified Variables

Quantified variables can take on a range of values, but only represent one at a time. They are used to build domain expressions in partial definitions for dimensioned shapes, although in theory they could be used in other ways. The quantifier expression gives the range of the quantified variable, and it is constrained to be of type integer.

The defblock in the quantifier is for a simple integer expression only, so it is a DataDecl, a complete definition, with no quantifiers, no local body nor multiple definition counter value.

```
encQuants :: Quantifiers -> CounterST [String]
encQuants [] = return []
encQuants (q:qs) = -- must be able to fold this ???
```

```
    do{ qss <- encQuant q; qsss <- encQuants qs; return $ qss++qsss }

encQuant :: Quantifier -> CounterST [String]
encQuant (QuantAll id) =
    do { elbl <- getnext
       ; return $ [ ("vdef('"++id++"',"++elbl++")")
                  , ("vdecl('"++id++"',t('Integer',[],[]))")
                  , ("qualuniv("++elbl++",all)") ]
       }
encQuant (QuantSome _ (DataDef id def _ _ _ )) =
    do { elbl <- getnext
       ; rngrels <- encDef id elbl def
       ; rnglbl <- getnext
       ; return $ [ ("vdef('"++id++"',"++elbl++")")
                  , ("vdecl('"++id++"',t('Integer',[],[]))")
                  , ("quantifier("++elbl++",all)") ]
       }

encQuant (QuantSome id _) = return [("error('"++id++"', invalid quantifier definition.)"
```

Operator Definitions

These are much like function definitions, but the function name is a symbol (wrapped in single quotes to make it a Prolog atom).

```
encOpDef (OperDef opsym fix def lbody lbl) =
  let lblsuf = if lbl > 0 then (show lbl) else "" in
    let opfname = case lbody of
                    { (LabelledBody id _) -> atomCombine id lblsuf
                    ; otherwise -> atomCombine opsym lblsuf
                    } in
      startCounterST opfname
        ( do { elbl <- getnext
             ; (rtlbl,drels) <- encDef opsym elbl def -- need to pull tts out of functio
--             ; let lbrels = encLBody lbody
```

```
--              ; return $ (("vdef("++(atomid opfname)++","++elbl++")"):(drels++lbrels))
             ; return $ drels
             } )
```

Complete and Partial Definitions

Encodes the defining expression of the identifier, returning the expression label, and the coded expression as a list. For partial definitions, it also constrains index expressions to be integers.

```
encDef :: Identifier->String-> Def -> CounterST (String,[String])
encDef _ elbl (CompleteDef je) =
    do { jerels <- encJoinExpr elbl je
       ; return $ (elbl, jerels)
       }
encDef _ elbl (PartialDef (IdIndex []) _) = return ("error("++elbl++", no index expressi
encDef id elbl (PartialDef (IdIndex dim@(e:es)) def) =
    do { let rank = length(dim)
       ; (dimslbl,dimsrels) <-
            do { d1lbl <-getnext
               ; d1rels <- encExpr d1lbl e
               ; results <-
                   foldM ( \ (is,rels) e ->
                     do { ilbl <- getnext
                        ; erels <- encExpr ilbl e
                        ; let const = "ec("++ilbl++",t('Integer',[]))"
                        ; return (is++ilbl, rels++(const:erels))
                        } )
                   (d1lbl,("ec("++d1lbl++",t('Integer',[]))"):d1rels) es
               ; return results
               }
       ; deflbl<-getnext
       ; (rtlbl,erels)<- encDef id deflbl def
       ; let arrd = "arraydef("++elbl++","++deflbl++",["++dimslbl++"])"
       ; let dtints = "dt(cocontpl("++(show rank)++",dt(interval(int))))"
```

```
        ; let vpc = "ec("++elbl++",t('Array',[tt(t('Top',[],[])),dt("++(show rank)++"),"+
        ; return $ (rtlbl, arrd : (vpc : (dimsrels++erels)))
        }
```

Function Definitions

For function definitions, generate a function interface definition, a validity check to see if the function interface matches the function class declaration and then the function implementation record, to be returned to encData function if the local body is provided.

The actual function implementation will be associated with a data identifier composed of the function name and the counter to make it unique; it will be constrained to have a Function type with the specified return and parameter types. This constraint is redundant to the validity check of the interface, but helps moves the mechanics of the CHR system along. The result expression will be constrained to it's return type; this is mandatory in a function definition, so it will be there.

The parameter expressions are all variable declarations, namely those of the formal parameters. Since at execution time their definitions must all be supplied by the function call, they are defined as input variables **vdef**(*parameter*,**cocoinput**(*parameter*)). This is identical to the method of managing definitions imported into a module from the importer, including possibly forcing a refinement of the input parameter types (but never outside of the subtype relationship).

```
encDef id elbl (FuncDef dbs je ft) =
    do { let n = length dbs
       ; relbl <- getnext
       ; (tlbl,trels) <- encFT ft
       ; let prels = concat (map encData dbs)
--       ; jerels <- encJoinExpr relbl je
       ; let pns = map idFromData dbs
       ; let pts = map typeFromDecl dbs
--       ; (pnlbl,pnrels) <- encParms pns
       ; ptlbls <- mapM (\t-> do{rs<- encFT t; return $ fst rs}) pts
       ; let ptlbl = foldl1 (\s lbl -> s++","++lbl) (wrap ptlbls)
```

```
--       ; let ecrel = "ec("++elbl++",t('Function',[tt("++tlbl++"),dt(int,"++(show n)++"
--       ; let rcrel = "ec("++relbl++","++tlbl++")"
       ; let funif = "funcintf('"++id++"',"++elbl++",tt("++tlbl++"),dt(int,"++(show n)++
                   "),["++ptlbl++"])"
--       ; let fund = "fnimp("++elbl++","++relbl++",["++pnlbl++"])"
       ; return $ (tlbl, (funif:trels))
       }  where
           wrap = map (\s-> "tt("++s++")")

encParms :: [Identifier] -> CounterST (String,[String])
encParms [] = return ("",[])
encParms (i:[]) =
  do { elbl<-getnext;
     ; return (elbl,["var("++elbl++",'"++i++"')","vdef('"++i++"',cocoinput('"++i++"'))"]
     }
encParms (i:is) =
  do { elbl<-getnext;
     ; let rels = ["var("++elbl++",'"++i++"')","vdef('"++i++"',cocoinput('"++i++"'))"]
     ; (mlbls,mrels) <- encParms is
     ; return (elbl++","++mlbls,rels++mrels)
     }
```

C.8.5 Attributes

Attributes are stored in a relation attr of the form attr(*class*, *val*) where *class* is the attribute class (e.g. range) and *val* is an expression defining the value of the attribute. In the current implemenation, the value is expected to be in the internal data and type term structures, like the dependent terms, because that is the only way to reason about them. When a Coconut interpreter is available, it will be possible to have the attribute values be any Coconut expression that can be resolved. If symbolic computation capabilities are added, then this could be stretched further. However, for now all attributes values must be expressable as booleans, integers, floats, or intervals or tuples of valid values.

```
encATs :: AttrList -> CounterST (String,[String])
encATs [] = return $ ("",[])
encATs (a:as) =
  do { (ats,arels) <- encAT a
     ; (asts,asrels) <- encATs as
     ; return (ats++asts, arels++asrels)
     }
encAT :: Attribute -> CounterST (String,[String])
encAT (Attribute ac val) =
  do { let edep = depExpr val -- encode value as an internal term
     ; return ("attr('"++ac++"',"++edep++")", [])
     }

-- for lists of full types, collect the string representation of each type
-- but concatenate the lists of extra relations generated.
encFTList :: [FullType] -> CounterST ([String],[String])
encFTList [] = return $ ([],[])
encFTList (t:ts) =
  do{ (tlbl,trels)<-encFT t
    ; (tlbls, tsrels)<-encFTList ts
    ; return  (tlbl : tlbls, trels ++ tsrels)
    }

encFT :: FullType -> CounterST (String,[String])
encFT (UnknownType) = return ("t('Top',[])",[])
encFT (VarType id supt) =
  do { (st,xsts) <- encFT supt
     ; return ("tv('"++id++"',"++st++")",xsts)
     }
encFT (GenType id sh as) =
  do { (shs,xshs) <- encSH sh
     ; (ats,xats) <- encATs as
     ; return ("t('"++id++"',["++shs++"],["++ats++"])", xshs++xats)
     }
```

```
encFT (NameType id deps as) =
  do { let dps = depExprList deps
     ; let ddps = dspList dps
     ; (ats,xats) <- encATs as
     ; return ("t('"++id++"',["++ddps++"],["++ats++"])", xats)
     }
```

Encoding Shapes

The shapes are encoded as lists of dependent arguments wrapped in the internal term representation. The name of the shape is already included in the type name.

```
encSH ScalarShape = return ("",[])
encSH (TupleShape fts) =
  do { (ets,xets)<- encFTList fts
     ; return ((wrap ets), xets)
     } where
         wrap = foldr (\e s-> ",tt(" ++ e ++ ")" ++ s) ""
encSH (DimShape des et) =
  do { let rank = show (length des)
     ; (dels,derels) <- encExprList des
     ; (det,erels) <- encFT et
     ; return $ ("tt("++det++"),dt(int,"++rank++")" ++ (wrap dels), erels++derels)
     } where
        wrap = foldr (\e s-> ",dt(interval(int)," ++ e ++ ")" ++ s) ""
encSH (FunctionShape ps rt) =
  do { let nps = show (length ps)
     ; (pts,prels) <- encFTList ps
     ; (rts,rrels) <- encFT rt
     ; return $ (("tt("++rts++"),dt(int,"++nps++")" ++ (wrap pts)), prels++rrels)
     } where
        wrap = foldr (\e s-> ",tt(" ++ e ++ ")" ++ s) ""
encSH (RecordShape flds) = return ("tt(t('Top',[]))",[]) -- records not done yet
```

C.8.6 Encoding Expressions

Each definition is a label for an expression. The expression may consist of:

- literal
- variable instance, possibly a field, array element or iterant, specifying a variable.
- unary operation on an expression
- binary operation on two expressions
- function application to a tuple of n expressions
- an expression in parentheses

Note that the variable instance is really an identifier instance. It is a variable if there are no extensions (e.g. x), but it could be a field (x.bound), an array reference (x[i]), or a function application (x(a,b)). Fields and records aren't currently supported.

Each expression is encoded as an individual constraint in type module. The expression is labelled with a generated symbol of the form *v_ennn*, where *v* is the identifier for the definition the expression is part of, and *nnn* is a variable length integer beginning with 0 for the main expression of the definition. Expressions will be numbered depth first in the syntax tree. There is no dependency built on this format, it is merely to assist tracing and debugging.

encExpr builds a list of constraints, as strings, while generating the unique expression labels. The numbering is handled using a state monad.

```
-- utility function to make a list of the form "[item1,item2,...,itemn]"
dspList :: [String]->String
dspList [] = ""
dspList ws = '[' : (foldr1 (\w s -> w ++ ',':s) ws) ++ "]"

-- ??? not handling joins and select statements yet ???
encJoinExpr :: String->JoinExpr->CounterST [String]
encJoinExpr elbl (NoJoin ce) = encCondExpr elbl ce
```

```
encJoinExpr elbl (Join _) = return ["error('"++elbl++"', conditionals not implemented)"]

encCondExpr :: String->CondExpr->CounterST [String]
encCondExpr elbl (Unconditional (e,et)) = encExpr elbl e
encCondExpr elbl (Conditional _) = return ["error('"++elbl++"', conditionals not impleme
-- ??? encoding expression type annotations ???

-- getting new labels for each expression, return in same order
encExprList [] = return $ ([],[])
encExprList (e:es) =
  do{ lbl <- getnext  -- get next expression label
    ; ers<-encExpr lbl e
    ; (lbls,esrs)<-encExprList es
    ; return $ (lbl:lbls, ers++esrs)
    }

numformat ft it val =  "t(" ++ ft ++ ",[],[attr('Range',dt(interval(" ++ it ++ "),("
                            ++ val ++ "," ++ val ++ ")))])"
encExpr :: String->Expr -> CounterST [String]
encExpr id (Factor (Lit l)) =
   return $ ["lit("++id++","++ ft ++ ")"]
     where
        (it,ft) =
          case l of
            LitInt i -> ("int", numformat "'Integer'" "int" (show i))
            LitReal r -> ("float", numformat "'Real'" "float" (show r))
            LitBool b -> ("bool", "t('Boolean',[],[])")
            LitString s -> ("string","t('String',[],[attr(length,dt(int,"
                             ++ (show (length s)) ++ "))])")
            LitSpecial Infinity -> ("cocoinf", "t('Top',[],[])")
            LitSpecial NegInf   -> ("coconeginf", "t('Top',[],[])")

-- currently only expands id extensions one level, e.g. can't have arrays of arrays
-- easy enough to fix
```

```
encExpr s (Factor (Var v [])) = return $ ["var(" ++s++ "," ++v++ ")"]
encExpr s (Factor (Var v (ext:exts))) =
    case ext of
      IdField _ -> return $ ["error('"++s++"'," ++ (show (Var v (ext:exts))) ++
                             ",Used field syntax, not yet supported)"]
      IdFunc ps ->
        do
          pls <- mapM (\_ -> do {n <- getnext; return $ n}) (replicate (length ps) v)
          let spls = dspList pls
          prs <- zipWithM encExpr pls ps
          return $ ("fnapp(" ++ s ++ "," ++ v ++ "," ++ spls ++ ")"):(concat prs)
      IdIndex ps ->
        do
          pls <- mapM (\_ -> do {n <- getnext; return $ n}) (replicate (length ps) v)
          let spls = dspList pls
          prs <- zipWithM encExpr pls ps
          return $ ("arrayref(" ++ s ++ "," ++ v ++ "," ++ spls ++ ")"):(concat prs)
encExpr s (UnOp f e _) =
    do l1 <- getnext
       s2 <- encExpr l1 e
       return $ ("unop("++s++",'" ++ f ++ "'," ++ l1 ++")") : s2
encExpr s (BinOp f e1 e2) =
    do l1 <- getnext
       l2 <- getnext
       s2 <- encExpr l1 e1
       s3 <- encExpr l2 e2
       return $ ("binop("++s++",'" ++ f ++ "'," ++ l1 ++","++l2++")") : (s2++s3)
```

Bibliography

[1] Wikipedia: Lattice(order).

[2] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. 2005.

[3] Ralph Beckett. *Mercury Tutorial*. Melbourne, Australia, w.i.p. edition, 2005.

[4] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A type-safe embedding of constraint handling rules into haskell. Master's thesis, School of Computing, National University of Singapore, S16 Level 5, 3 Science Drive 2, Singapore 117543.

[5] B.A. Davey and H.A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition edition, 2002.

[6] Thom Frühwirth. Theory and practice of constraint handling rules, 1994.

[7] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Dept. of Computer Science, Universiteit of Nottingham, University Park, Nottingham NG7 2RD, UK, 1996.

[8] Haskell Workshop 2005. *Putting Curry-Howard to Work*, 2005.

[9] C.B. Jay. *The FISh language definition*. School of Computing Sciences, University of Technology, Sydney, P.O. Box 123, Broadway NSW 2007, Australia, 1998.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[11] P.J. Stuckey and M. Sulzmann. A systematic approach in type system design based on constraint handling rules, 2001.

[12] P.J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[13] Martin Sulzmann. *A generla Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Dept of Computer Science, 2000.

[14] Martin Sulzmann, Martin Muller, and Christoph Zenger. Hindley / milner style type systems in constraint form. Research Report ACRC-99-009, University of South Autstralia, School of Computer and Information Science, 1999.