The implementation of a Portable Software Platform

A dissertation for the degree of Doctor of Philosophy

Carl David Sutton
March 18th 1994
Acknowledgements

For professional guidance and assistance, I would like to thank my supervisor, Chris Jesshope; and also Slava Muchnick and Sasha Shafarenko. For helping to maintain my solvency, I am very grateful to SERC, and PARSEQ limited.

For seraphic personal support, guidance, and assistance, I am very grateful to my parents and step-parents; to Peter Sealey and Pauline Cass; and to Beatrice Fitzgerald.
Abstract

The implementation of a Portable Software Platform

Traditionally, languages were created and intended for sequential machines and were, naturally, sequential languages. All that was required to maintain portability between machines were full specification compilers. These languages have not adopted well to the parallel domain since they are poor at expressing extractable concurrency. Portability is obviously as important for parallel computers — but the proliferation of ad hoc languages for parallel computers indicates that portability is not always the prime consideration of the language developer, as it should be, but that the support for a particularly specific un-general-purpose parallel computer is.

A Portable Software Platform (PSP) is an intermediate level for compilers for parallel and scalar machines; the particular PSP discussed in this thesis, F-code, is a PSP for imperative, computational programming languages. As an intermediate level, the PSP must be general enough to represent all high-level programming problems, without discarding explicit, known concurrency; the PSP must be able to infer what other parallelism exists; and it must also be general enough to support all general-purpose parallel machines.

The underlying bases of computational programming languages for parallel computers are data-parallelism and functional concurrency. Data-parallelism should therefore be represented in a PSP in the most descriptively simple, and hence most manipulable, way; and also, data-parallel operations are evaluated lazily — which means that only those elemental computations which have any bearing on the result of a computation are done.

As a very general representation medium for computation, a PSP must also be architecture-neutral: PSP programs must be compiled efficiently to all general purpose parallel machines. A specific, machine-dependent, implementation must be inferred from an architecture-neutral PSP program, making it match specific aspects — memory and arithmetic pipelining, data-partitioning, VLIW execution — of a parallel or scalar hardware platform. Descriptive simplicity means that F-code is very suitable for data-parallel optimization: a PSP can equally be thought of as a tool for data-parallel optimization.
Architecture-specific aspects are deferred to the very last stages of the compilation process of a PSP. The particular implementation of an F-code compiler given in this thesis is arranged in a number of main stages: the front-end is architecture-neutral; the code generator is generalized for the class of RISC processors; and only the very last stage of the compiler (the targetter) requires any specific details of a particular RISC processor. Thus, not only is a PSP architecture-neutral, but the process of compiling PSP programs maintains architecture-neutrality to some degree as late on into the compilation as possible.
# 4 Sequential aspects of F-code

## 4.1 Introduction

## 4.2 Identifying lazy environments

## 4.3 Lazy evaluation across index spaces

### 4.3.1 Optimizing data-parallel loops

## 4.4 Indices

### 4.4.1 Notation for indices

### 4.4.2 Scopes of indices

### 4.4.3 Shape trees and indices

### 4.4.4 Folded shape trees

## 4.5 Sequential ordering of an F-code tree

### 4.5.1 Example of coarse-level sequentialization

### 4.5.2 Comparison between lazy environments and I-structures

### 4.5.3 Pipelining between almost adjacent lazy environments

## 4.6 Summary and conclusions

## 4.7 Applying this chapter to parallel machines

---

# 5 Code generation

## 5.1 Introduction

## 5.2 A machine-independent graph for targetting

### 5.2.1 F-code types, machine types

### 5.2.2 Abstract machine instructions

### 5.2.3 Control dependency arcs

### 5.2.4 Data dependencies

### 5.2.5 Integrating control dependencies into a data dependency graph

### 5.2.6 Traversing the graph

### 5.2.7 The representation of loops

## 5.3 Transforming F-code into a T-code graph

### 5.3.1 Lazy evaluation and lazy-environments

### 5.3.2 Side-effects

### 5.3.3 Scopes of single-assignment variables

### 5.3.4 Loops

### 5.3.5 Indices and implementing geometric operations

### 5.3.6 Examples

### 5.3.7 Blocks, memory and stacks
6 Targetting

6.1 Transforming T-code into machine instructions

6.1.1 Primary rewrites \( p \) and \( r \)

6.1.2 Transforming loops

6.1.3 Transforming ifs

6.1.4 Condition codes

6.1.5 External calls and register coalescing

6.2 Inference of addressing modes

6.2.1 Immediate constants

6.2.2 Unavailable addressing mode

6.3 Giving the graph a spine (sequentializing)

6.4 Iterative scheduling and the current implementation

6.4.1 Execution pipelining

6.4.2 Arithmetic pipelining

6.4.3 Superscalar scheduling

6.4.4 Memory pipelining and the cache

6.4.5 Register allocation and register coalescing

6.4.6 An iterative approach to low-level scheduling

6.4.7 The initial implementation of the targetter

7 Conclusions, contributions, and future work

7.1 Conclusions

7.2 Future areas of work

7.3 Contributions

A Definition of F-code

A.1 Objects

A.2 Representation of the F-program

A.3 Identifiers

A.4 Literals

A.4.1 Constants

A.4.2 Operators

A.4.3 Keywords

A.4.4 Function Labels

A.5 Evaluation of the F-program

A.6 Functions

A.6.1 const

A.6.2 hold

A.6.3 template
A.6.4 create ................................................................. 216
A.6.5 var ............................................................... 217
A.6.6 select ............................................................. 218
A.6.7 global .............................................................. 219
A.6.8 dispose .............................................................. 219
A.6.9 local ............................................................... 219
A.6.10 mark ............................................................... 220
A.6.11 ramp ............................................................... 220
A.6.12 monadic ............................................................ 220
A.6.13 dyadic ............................................................ 221
A.6.14 choice ............................................................. 221
A.6.15 reduce ............................................................. 221
A.6.16 transp .............................................................. 222
A.6.17 sect ............................................................... 222
A.6.18 slice ............................................................... 223
A.6.19 repl ............................................................... 223
A.6.20 pack ............................................................... 223
A.6.21 gather ............................................................. 224
A.6.22 diag ............................................................... 224
A.6.23 transform .......................................................... 225
A.6.24 part ............................................................... 225
A.6.25 comp .............................................................. 225
A.6.26 pol ................................................................. 226
A.6.27 displace .......................................................... 226
A.6.28 distance .......................................................... 227
A.6.29 type .............................................................. 227
A.6.30 shape ............................................................. 227
A.6.31 assign ............................................................. 228
A.6.32 channel ........................................................... 228
A.6.33 put ............................................................... 228
A.6.34 get ............................................................... 229
A.6.35 seq ............................................................... 229
A.6.36 comma ........................................................... 229
A.6.37 par ............................................................... 229
A.6.38 loop ............................................................... 230
A.6.39 spawn ............................................................ 230
A.6.40 if ................................................................. 230
A.6.41 call ............................................................... 231
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B  Lexical table</td>
<td>233</td>
</tr>
<tr>
<td>C  Parse table</td>
<td>237</td>
</tr>
<tr>
<td>D  Type inference table</td>
<td>241</td>
</tr>
<tr>
<td>E  Type inference rules</td>
<td>245</td>
</tr>
<tr>
<td>F  Abstract machine instructions</td>
<td>251</td>
</tr>
<tr>
<td>G  Targeting the i860 from T-code</td>
<td>259</td>
</tr>
<tr>
<td>Bibliography</td>
<td>265</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Portable software platforms ................................................................. 10
1.2 General overview of the F-code system ............................................. 18

2.1 Type compatibility ................................................................................ 32
2.2 The geometric operation sect ................................................................. 33
2.3 The geometric operation diag ................................................................. 34
2.4 F-code compose operator ..................................................................... 34
2.5 The F-code operator slice .................................................................... 35
2.6 Context positions .................................................................................. 37
2.7 A general data-parallel assignment ...................................................... 38
2.8 Orientation for diag ............................................................................. 41
2.9 The reduce function ............................................................................ 42
2.10 Shape coercion .................................................................................... 43
2.11 Lazy and eager sections ..................................................................... 47
2.12 Example of lazy and eager sectioning ................................................ 48
2.13 Architecture-independent compiler stages ......................................... 53
2.14 Architecture-dependent compiler stages ............................................ 58

3.1 Front end compiler phases ................................................................. 59
3.2 Example of table-based parsing ......................................................... 64
3.3 The aim of shape inference .................................................................. 71
3.4 Shape expression created for ramp ..................................................... 77
3.5 Shape expression created for dynamically shaped ramp ...................... 78
3.6 Lazy environments created for dynamically shaped ramp ................... 79
3.7 Permitting parallel function invocation .............................................. 87
3.8 slice of sect / sect of slice .................................................................. 89

4.1 Sequentialization compiler phases ..................................................... 97
4.2 Subordination of a sequential compiler in a distributed one .............. 98
4.3 Identification of lazy environments .................................................. 99
4.4 The shape inference for lazy evaluation of reduce ........................... 101
4.5 The indices for lazy evaluation of reduce .......................................... 102
6.7 Iterative scheduling ................................................................. 187

7.1 Layers of an F-code compiler .................................................. 197
Chapter 1

Introduction

This thesis is about the implementation of an F-code compiler. F-code is a Portable Software Platform (PSP) for imperative programming languages. A PSP is an intermediate level interface for compilers of high-level languages for parallel machines. A PSP decouples high-level languages from parallel machines. A PSP is a single standard which integrates compilation and optimization techniques for a wide variety of languages and machines.

In section 1.1, this introductory chapter first defines the three terms architecture-specific, architecture-independent and architecture-neutral, which are used to differentiate between good and bad platforms. A PSP must be architecture-neutral, meaning that it is an applicable intermediate level interface for any number of high level languages, and all general purpose machines. In section 1.2, the introduction continues by describing the types of high level language which are currently in use, and the trends in programming languages. In section 1.3, the term Portable Software Platform is defined, along with a justification of the need for PSPs. Section 1.4 describes some of the features of parallel machines a PSP must accommodate. Finally, section 1.5 describes some contemporary compilation techniques used in conventional parallelizing compilers. Section 1.6 describes the general aim of this thesis. F-code is defined in chapter 2.
1.1 Introduction of terms

The incongruity between languages and parallel architectures

There are a number of sources of parallelism in programming languages which can be utilized by compilers for parallel machines: data-parallelism, functional-concurrency, process concurrency. The relative importances of these depends on the application. Limiting the discussion to the main requirements of scientific users, data-parallelism is the most significant. For execution on parallel machines, non-scalar data may be expressed by combinations of sets, tuples, lists, arrays or, in irregular problems, graphs, etc. Most languages may represent or implement these structures in some way: C is perfectly capable of representing arbitrary data structures although it is nothing more than an assembly language. The important question is: can enough parallelism be extracted from a program to permit a compiler to produce executable code congruent with a parallel machine architecture? This depends on the capability of the programming language to represent known parallelism in an extractable way and also on the capability of the compiler to re-extract it and to infer what other parallelism exists. Obviously then, C, while a general enough programming language, is not congruent to parallel machines.

The incongruity between ideas and languages

A programming problem, a parallel algorithm, independent of the language in which it is to be coded, independent of the machine on which it is to be executed, is language-neutral and architecture-neutral: it is not developed with a language or architecture in mind at all. While the problem's parallelism may be conceptualized by the programmer, the language in which the problem is implemented is always to some extent restrictive and therefore codings are always dissonant to the required meaning: while the functional specification of a problem can be adequately coded in almost any language, it is not always possible for the compiler to infer the inherent parallelism of the algorithm. It is most often impossible for the programmer to specify what parallelism is known; and when it is not known or cannot be expressed, it is most often impossible for the compiler to infer it. This is backed up in [HA90] which shows examples of parallelism in LISP programs that are overlooked by parallelizing LISP compilers, as well as parallelism which can be uncovered by compilers which difficult or impossible for users to express.

There is a requirement for explicit parallelism in programming languages, while the compiler infers other implicit parallelism. If a language does not have 'intrinsics' for parallel operations on arrays, computation on arrays cannot be represented in the most descriptively simple way, and hence parallelism in the most extractable way. In this case, the quality of the implementation overwhelmingly depends on the quality of the vectorizer [Wol82]: it is common to still use scalar languages like FORTRAN 77 and vectorize programs to execute on vector machines, but much

---

1 Congruence: 1. the level of ability of a language, or code, to make efficient use of architectural features of a machine [Ski91]; 2. the level of ability of a language to represent a problem in an architecture-neutral way. Congruent: possessing a high level of congruence.
1.1. **INTRODUCTION OF TERMS**

less parallelism can be extracted from an application than actually exists [AR90]; indeed much less than may have been known to exist. While it is nonsensical to discard known parallelism and re-vectorize a program to execute on a parallel (vector) machine, due to commercial inertia (the inability to adopt new languages) this is what really happens. It is also common practice to expect programmers to restructure FORTRAN 77 loops in a program to conform to a set of rules for 'vectorizability' [Ken92] sometimes interactively, aided by the compiler. It might also be possible to expect programmers to restructure FORTRAN 77 code to comply with sets of rules for 'concurrentizability' (for parallel machines in general) but fortunately the complexity of parallelization is prohibitive. Another restriction of languages is, for example, if the language cannot extract parallelism from graphs used to represent irregular problems but still does have a congruent implementation of arrays. In this case, the computation can often be mapped onto arrays, but many of the dependencies are hidden in indirect array references causing a loss of potential parallelism due to necessarily conservative compilation; and the implementation is not as congruent as it might be. Run-time compilation [SBW90] might alleviate this in simple cases.

**The effect of losing architecture-neutrality**

It has been shown that any problem which is architecture-neutral when coded in a language which is inadequate at specifying the parallelism of the problem may lose a certain amount of explicit parallelism because compilers cannot re-infer it. This is true for coding in any language since languages and compilers are still primitive. But architecture-neutrality is what is really required by programmers: an ability to specify the problem and known parallelism clearly, disregarding the underlying machine (thereby portably), and the ability of the compiler to provide an efficient implementation. Architecture-neutrality basically means portable efficiency and language neutrality.

**Classifying the mismatch between idea and architecture**

Architecture-neutrality is the perfect case. When it comes to coding a problem, if a language is 'reasonably congruent' with any current general-purpose machine, but not necessarily congruent to the problem, it is architecture-independent. This is weaker than architecture-neutral, where congruence with the problem is also important. 'Reasonably congruent' is hardly a scientific term, but is best described by showing what is not reasonable at all.

Many languages are tailored to suit particular machines because of a pragmatic, commercial, requirement for ad hoc efficiency with ease: For example: message passing machines may use a dialect of FORTRAN with message passing primitives, or a vector machine may use vector languages like EVA [DMP90]. The most common language for supercomputers is FORTRAN and in order to take advantage of specific features of particular machines, there are many dialects of FORTRAN in current use, 12 of which are compared in [KB88]. The problem with these dialects is that because they are tailored to particular machines or classes of machines, they
are architecture-specific: programs written in these languages are not portable because they cannot be efficiently implemented on any but the machine they were intended for. The are only congruent to individual machines, and not ‘reasonably congruent’ in other cases.

There is thus a hierarchy of terms:

— **Architecture-neutral:**
  
  The abstract formalization of the problem.

  *Congruent to both idea and any general-purpose architecture.*

  Congruence with the idea is attained by using declarative semantics. Abstracts away from any specific architecture altogether: the formalization of the problem gives a definition of an abstract machine which may be implemented on real machines. Congruent implementations for a particular architecture are inferred from the architecture-neutral specification. Optimization may be guided by inferrable congruence. Compilers must produce code of comparable efficiency to architecture-specific compilers.

  Examples: λ calculus [Chu41], Bird-Meerten Formalisms [Bir87, Bir89], F-code [BMS92].

— **Architecture-independent:**

  The concrete but portable expression of the problem in a high-level language.

  *Reasonably congruent to any general-purpose architecture.*

  Regardless of the general ability of the language to specify the problem’s inherent parallelism and the compilers ability to re-infer such parallelism, it can be compiled with roughly equal, apparently optimal, efficiency on all general purpose machines. The language abstracts away from underlying communication on distributed machines, etc. to a sufficient extent to make it portable.

  Examples: FORTRAN 90 [For91], HPF [HPF92], FORTRAN D [PHK+92], Vienna FORTRAN [ZBC+92].

— **Architecture-specific:**

  The concrete but not portable expression of the problem in a language.

  *Congruent only to a specific machine or class of machines.*

  Examples: C, FORTRAN 77, DAP FORTRAN, PCF FORTRAN [PCF88], Cedar FORTRAN [EHJP90].

This should make the difference between architecture-neutral, -independent and -specific apparent. Architecture-independence is somewhat weaker than neutrality, usually taken to mean: ‘general enough to model several architectures’ [Ski91]. It should mean ‘model all general-purpose architectures’; it is impossible to say just ‘all architectures’ since some are application-specific, or it would be easy to invent a machine to break the rule. The expectation for supercomputer manufacturers is therefore to produce machines which are general-purpose. SIMD computers are not general-purpose: since they are single-threaded machines, they can not efficiently execute
process parallelism in a spatial sense. It would be very difficult to do graph reduction, and hence execute λ calculus, on a SIMD machine efficiently.

Architecture-neutral languages are the most abstract; architecture-specific languages are primitive, ad hoc languages which expose a great deal of the machine's architecture to the user. An architecture-independent language is one which assumes a certain fixed abstract machine. For architecture-neutrality, the abstract machine is defined by the problem and made congruent to a real machine by optimization. An architecture-neutral formalisation is unlike a PRAM model [MV84] where the model may be emulated by real machines with a certain regular loss of efficiency, but is rather a mapping from a problem-specific abstract model to a congruent, assembly-language-level implementation on a real machine.

F-code is the first practical architecture-neutral platform

λ calculus is classified as architecture-neutral. That means that it is very abstract, and can be implemented on any general-purpose machine reasonably efficiently. It is best implemented on tightly coupled MIMD systems, not SIMD machines. Considering this, λ calculus could therefore be classified as architecture-specific. It is seems more accurate to say that SIMD machines are not general-purpose. Bird-Meerten formalisms and F-code are applicable to SIMD machines; λ calculus is not.

The congruence of λ calculus to a machine is difficult to know [Ski91]: “what congruence ought to mean in a functional language setting is difficult to know because of the existence of semantic-preserving transformations that can dramatically change the execution cost of a computation. Even if one postulates the existence of an optimal transformation system and defines execution cost for functions there are still substantial overheads in current implementations of graph reduction. Whether these are inherent remains an open question.” Also λ calculus does not support assignment directly, and so λ calculus, as is, cannot be used as an intermediate platform for imperative programming languages.

Later in this introduction it will be shown that there is a flaw in the Bird-Meerten formalism which limits its usefulness in practical implementations.

This leaves F-code as the first and only example of a practical architecture-neutral platform, for imperative programming languages. The definition of F-code is included in appendix A; its implementation is described throughout this thesis. This thesis describes the first implementation of an architecture-neutral platform.

1.2 High-level languages

Categorizing languages

There are roughly two categories of parallel languages: (a) imperative and (b) declarative. Imperative languages are based on the more classical sequential Von Neumann paradigm including
assignment. An imperative program can be seen to be an ordered system of assignments, or a system of state transitions — the program can be seen to have state. Declarative languages (viz. logic and functional/applicative programming languages) generally use a notation to describe what is to be calculated rather than how it is performed or implemented: declarative semantics. Declarative languages are generally stateless. In the declarative category, functional programming languages such as SASL [Tur76], KRC [Tur81], Miranda [Mil85], Haskell [HE88] are based on functions and logic programming languages such as Parlog [CG86] and Concurrent Prolog are based on relations.

Imperative languages

The most prevalent type of language are imperative languages (procedural or object-oriented languages). These types of language are quite naturally augmented with parallel features: either process-parallelism or data-parallelism.

Object oriented languages like C++, which is becoming increasingly popular, are often extended to include data-parallelism and process parallelism: Examples of this type of language are Concurrent Smalltalk [YT86] and Emerald [BHJ+87]. In object-oriented languages, data-types and the operators that may operate on them (collectively the class) are objects, and each active object can be an active process. Parallelism can be introduced around asynchronous message passing between objects, broadcasting, and also having data-parallel objects. Objects interact only through message-passing which invokes operations in the class of an object.

Most computational programs are written in languages like FORTRAN. Purely sequential languages were created and intended for sequential machines; full specification compilers were all that were required to maintain portability of code between different scalar machines; however they have not adopted well to the domain of parallel machines.

Parallel computers are remarkably difficult to program, especially with the wrong types of language. The use of the wrong types of language is exemplified by programmers having to write programs in languages (or dialects) which are architecture-specific: FORTRAN has message-passing dialects for MIMD distributed-memory machines; array-syntax dialects for SIMD machines; and explicitly parallel dialects, with synchronization, for MIMD shared-memory machines. It is useful to note that codes written in these dialects are not portable between vendors' machines and often do not even perform well across different products from the same vendor.

More recently, standards have been proposed, and partially implemented, to augment procedural languages with array syntax, alignment, and distribution mechanisms, to make them more general-purpose: FORTRAN 90, FORTRAN D [FHK+92], Vienna FORTRAN [ZBC+92] and HPF [HPF92].

Array extensions — array notations, forall loops, extra intrinsics — all attempt to make vectorization easier, or unnecessary — to make message-passing for distributed machines implicit. Sequential languages have an inability to express data-concurrency in an extractable
form; these new standards introduce extensions (maintaining backwards compatibility) to FORTRAN to do just that and to partially guide data-distribution. FORTRAN 90 came from FORTRAN 8x [Rei87] which introduced whole array expressions and assignments, array sections, where statements, and a good deal of intrinsical functions, including dot-product, matrix multiplication, reductions, array constructions, and shifts, transpose, etc. These are quite typical for array extensions to conventional FORTRAN 77. All of these languages intend to be architecture-independent.

If one considers the abstractions these languages provide for data-parallelism, the most general well known language is APL (and APL2.) For example: 

\[
(+/\{1\} 3 4 p = 12) \text{ creates a 12 element vector containing a ramp } (p=12), \text{ reshapes it into a } 3 \times 4 \text{ matrix } (3 4 p), \text{ then takes the add-reduction } (+/) \text{ along axis 1 } (+/\{1\}) \text{ to produce a four element array: } (15 18 21 24).
\]

This is quite functional, and some of these ideas are integrated into the functional language MOA [Mul88], which is described more later, but APL is an imperative language since it includes [data-parallel] assignment: \(AT \leftarrow 3 5\) is equivalent to the scalar operations: \(AT + \leftarrow 3\) \(AS \leftarrow 5\). It also includes selective assignment (overwriting only parts of an object which are specified), indexing, and a general set of data-parallel operations; operands to operators may also be functions themselves. Matrices and higher-order arrays can be reshaped from lists and vice versa. APL cannot be compiled but may be interpreted in hardware [HHSS92].

A good formalism for manipulating data-parallel objects provides a higher level of congruence with the ‘idea’ of a program — thus aiding architecture-neutrality. APL, however, does not possess a type-system; a PSP must, of course, possess a rigid type system, such that a program can be inferable to be monomorphic, and thereby be compiled to execute efficiently. The congruence of an implementation with a machine depends on the ability of the compiler to infer a monomorphic implementation, rather than cloning code for different types and inserting run-time checks for type.

**Functional languages**

The difference between functional languages and imperative languages is not all that great, as noted in [Hud89]. The expression is an important concept of languages like FORTRAN, with a strong mathematical basis. Subsequently Pascal built on the theme. Functional programming languages can be regarded as the logical extension of the trend, where everything is a function/expression. They differ because of assignment, but functional parallelism has an important part to play in imperative language implementation for expressions that do not use side-effects. Functional parallelism can be used to hide latency in an application with excess parallelism [BMS92]. Side-effects are the important omission in functional programming languages such that expressions are guaranteed referential transparency [PJ88, FH88].

Functional programming languages ought not be discussed in abstracto without mentioning their manners of execution: generally λ calculus or data-flow (static [Den85] or dynamic [GW80].)
A calculus [Chu41] is the basis of many functional languages, the first of which was LISP [McC60, McC63]. A calculus is a very simple but very general computational model devised by Alonzo Church in the 1930s which may be evaluated by graph reduction either in applicative or normal order (effectively call-by-value and call-by-name respectively). The important part of the calculus for parallel machines is the Church-Rosser property: the same answer is produced when reducing any expression regardless of reduction ordering. Higher level functional languages may be constructed from λ expressions and the syntax is called syntactic sugaring because although the representation of the language is changed, the underlying meaning stays the same. A finite number of substitution steps results in simple λ calculus rules for evaluation. Therefore only simple λ calculus needs to be executable. λ calculus is architecture-neutral, but unfortunately, as it stands, it cannot support assignment.

It has generally been thought that assignments destroy referential transparency and require a determinate evaluation order. However in [ORH93], an extension is given to λ calculus to represent additional constructs and reduction rules that represent modifiable variables and assignments. The calculus has the Church-Rosser property. In other words, it is possible for a functional language to include assignments and mutable variables: a language may naturally express advanced imperative constructs without destroying the algebraic properties of the functional subset.

In general, declarative programming languages are more inefficiently executed than imperative ones since they require a certain overhead: functional languages have overheads due to lazy evaluation and higher-order functions (functions passed as parameters). So even augmented with assignment [ORH93], λ calculus is not a good PSP for imperative languages. Lazy evaluation, however, in data-parallel programs — evaluating only those atomic parts of a data-parallel operation which affect the result — brings large efficiency gains. If data-parallel operations can also be curried, thereby combining lazy evaluation of operations, the efficiency gain can be even more marked. Thus lazy evaluation and a kind of currying are adopted in the implementation of F-code. This will be shown in later chapters.

Logic languages

Lastly, logic languages, the first of which was Prolog [CM87], developed in the 1970s, attempt to prove clauses and create entities in bound variables as side-effects of the proof. Prolog is primarily used in the field of artificial intelligence. It is possible to introduce parallelism to Prolog in three main ways: AND-parallelism, OR-parallelism and by parallelizing the unification algorithm.

Strand [FT90] is a single-assignment language, where variables provide the abstraction for message-passing and synchronization. When a value is assigned to a variable, it is automatically available to all other processes that share the reference, transferring the data to the required locations and hence also providing synchronization. A typical clause includes one or more guard components and a number of bodies. Similarly to Flat Parlog [FT87], guards cannot call user-
defined predicates (which gives the program its "flatness"). Like CSP [Hoa85], it is arranged like a network of processes, waiting to be reduced, with variables providing the links. Thus its main execution model is a process model, and main source of concurrency is process concurrency.

Since the aim of F-code is to support data-concurrency, logic languages are not explicitly supported. But F-code includes the notion of channels, "PAR", and "SEQ", and so process-concurrent languages like OCCAM [Occ84] are supported.

Standardization

The trend in high-level languages is to include primitives for explicit data-parallelism: languages like FORTRAN 90 include array notations and a certain amount of intrinsical functions to operate on arrays. What underlie all parallel languages for supercomputers is data-parallelism whether it is to be extracted by vectorization, or represented explicitly.

But besides the trend, there is no consensus regarding what $U$ should be like, and there are a growing number of parallel languages. Fortunately, there are a number of efforts to standardize programming languages [Per92]:

1. The Parallel Computing Forum are working on FORTRAN extensions for programming shared memory parallel computers [PCF88]. This is architecture-specific. This was passed to ANSI subcommittee X3H5 for which compilers will not be made available until a final standard has been approved.

2. The High Performance FORTRAN Forum are working on extensions for FORTRAN, for a language called HPF [HPF92]. It is an architecture-independent language. Its extensions for FORTRAN intend to support data-parallelism and top performance on parallel computers with non-uniform memory-access costs. Its definition is not sanctioned by any national standards body, but several vendors including DEC, Intel, MasPar and Thinking Machines have promised HPF compilers and so it is expected to be a de facto standard [Koe93].

3. In the functional language area, the Haskell language [HE88] is a standardization of the work of a group of researchers from Europe and the USA.

It is debatable whether parallel programming languages are mature enough to be standardized; however a vast proliferation of languages is undesirable. Standardization is not only of the language itself, but also of the libraries and tools available for a computer.

A pessimistic note is given in [Pan92]: "Given the diversity of current language implementations and conflicting needs within the parallel programming community, will it be possible to devise a single standard that satisfies the demands? Not in the near future.".
1.3 Portable Software Platforms

Burton Smith in a recent keynote speech [Smi90] said: "A general purpose supercomputer should provide a machine-independent programming environment." This sentiment is widely accepted in principle, and vital in practice, in order that parallel machines can be accepted by general users: programmers are obviously reluctant to rewrite applications for every new machine [Ken92]. The aim of the programming environment is to hide architectural details of machines and thereby make an application portable between machines. Its single aim is portable efficiency.

![Diagram of Portable Software Platforms]

The purpose of portable software platforms

The purpose of the programming environment is to decouple the machine from high-level programming languages. A single programming environment supports a number of high-level languages. As a generalized programming environment, it must incorporate the imperative constructs and data-types representable by high-level languages. As a single standard, it can collect libraries of techniques for parallelizing compilers; collect optimization techniques; provide a means of automatic availability of different languages and paradigms on different machines without portability problems; and provide a fast turn-around for compilers for new languages (figure 1.1).
1.3. PORTABLE SOFTWARE PLATFORMS

It is an intermediate level for compilers, supporting reflexive optimization. High-level compilers to a PSP do not need to do much optimization because this is a particular strength of the PSP. Intermediate languages since UNCOL [Str58] have been used to write portable compilers for scalar computers. P-code among others was dedicated to Pascal compilers [Nel79], A-code to ADA [Dom80], etc. P-code was later augmented with vector extensions [Tur87]. But of course there intermediate languages are all machine or language specific.

Another facet of PSPs is their use in compiler generators [CC92]. PSPs are formally, rigourously defined and the meaning (the functional specification) of programs represented by a PSP is unequivocal. They are therefore suitable source languages for compiler generators.

Portable software platforms should be data-parallel

The programming environment, since it supports parallel machines, naturally ought to be parallel. Since the discussion is limited to scientific users, the programming environment ought to be data-parallel. There is a clear argument in favour of SPMD-style (data-parallel) programming languages [Ski91] which basically says: (i) process concurrency is difficult to program; even more difficult to debug (ii) data-parallelism provides a simply managed abstraction for programming parallel machines, even though it restricts the forms of computation which can be expressed to single-threadedness.

General applicability

There is an argument which says that parallel computers are not applicable for general use by scientific and other users until there exists a suitable compilation strategy. Valiant [Val90] expounds the fact that parallel computers have limited applicability until, as already for sequential machines, there exists a triple \((U, A, M)\) such that there is at least one high level language \(U\) that humans find satisfactory for expressing arbitrary, arbitrarily complex algorithms; a machine architecture \(M\); and a compilation algorithm to transform arbitrary programs in \(U\) to run on \(M\) efficiently. In the sequential domain there is the Von Neumann model; in the parallel domain there is nothing comparable. The subject of this thesis is \(A\) and its algorithms which will collectively be called a Portable Software Platform (PSP).

PSPs must be architecture-neutral

First of all, no satisfactory \(U\) exists. One can only say that \(U\), if it existed, would be at least architecture-independent and standardized. There is no single language for parallel machines suitable for coding every application. The situation is worse still: it is common practice to use several languages in development of a single application: For example: using C++ as a high-level language for modularity, FORTRAN as a high performance assembly language for coding computationally intensive fragments, and using AVS for visualization [PCRC93]. Also a user’s preference for any particular programming language has little to do with that language’s
particular merits. It most often depends on how widely available the language is and also on the availability of software libraries [MSS93]. There will always be more than one high level language. It is immediately obvious therefore that $A$ must support any number of high-level languages; $A$ must be language-neutral; a PSP must be language-neutral. A PSP must be architecture-neutral because it supports more than one architecture-specific or architecture-independent programming language.

The functional subset of PSPs

A PSP program must be some functionally-identical representation of a program written in any high level language and the PSP must be reached in a finite number of substitution — perhaps pattern-matching and transformation — steps from the high level program.

A PSP can contain a functional subset, which improves congruence with the problem, and include assignment. A PSP must be able to support assignment since it is a platform for imperative programming languages. As already described, $\lambda$ calculus cannot be used as a PSP, as is, because it cannot support assignment. Theoretically, [ORH93] introduces imperative assignment to $\lambda$ calculus and this work is the closest known equivalent to F-code, except F-code has intrinsical data-parallel features which can be used for optimization.

The definition of a PSP

A PSP is a practical model for parallel computation. A PSP is a programming language that possesses:

1. **High-level semantics** including all sorts of abstractions inherent in high-level languages such as tuples, lists, pointers, data types (including arrays), etc.

   In a recent study [MS89], the requirements for useful parallel data structures were enumerated: Data structures should: (1) match existing and future computer architectures; (2) allow for efficient parallel implementations; (3) be formally defined in an applicative language; (4) allow the definition of complex objects in a constructive way.

   Also, perhaps, some properties of object-oriented languages are necessary: perhaps just simple classes [Pan92]. Most importantly, the PSP must have a type system that is interprocedurally verifiable as type-safe at compile time.

2. **Primitive syntax** making it easier for the top-level compiler to generate code. What needs to be expressed are data-dependencies at the data-parallel (SPMD-style) level; the syntax may be inconvenient because a human will never need to use it. It must possess descriptive-simplicity [Ski91] reducing the overhead of describing and managing massive-parallelism. Three kinds of abstractions exist for descriptive-simplicity: (1) abstractions for decomposition or the explicit expression of parallelism; (2) abstraction from the details of communication; (3) abstraction from the details of synchronization.
1.3. PORTABLE SOFTWARE PLATFORMS

Descriptive simplicity is also aided by omitting all but necessary type information from the notation, and using a type-inference procedure: what is not present can be inferred. Also, since the computation is a data-parallel one, objects and operations have shape ( extents), which as much as possible may also be omitted from the notation. Objects must be able to be of any dimensionality since this is imperative for language-neutrality. Descriptive simplicity is aided therefore by using a shape-inference procedure to infer the shapes of objects and operations. It is necessary to know the shapes of objects and operations in order that the representation may be compiled.

3. Architecture-neutrality. There should be no assumption about specific hardware features (like the addressing range, machine data formats and alignments, etc.) Architecture-neutrality also implies declarative semantics: the PSP program represents what is to be computed rather than how.

A PSP is a virtual machine, which does not enforce any programming methodology. It is not a low-level specification of the algorithm.

Since it is architecture-neutral, the compiler must be able to infer congruent distributions and redistributions for efficient parallelized execution on distributed-memory machines.

Current state-of-the-art parallelizing compilers cannot do automatic-parallelization; it is sufficiently complex to be thought to be an impossibility. High level languages include distribution directives (which are machine-specific), or the MIMDizer [MIM91] and SUPERB [ZBG88] are interactive semi-automatic systems. An architecture-neutral platform can be interactive (since data-partitioning is handled interactively), and this is not inconsistent with architecture-neutrality, but it should not include directives — because they (or the choice of which to use) are inevitably machine-dependent. This problem is out of the scope of this thesis, since this thesis is about the implementation of F-code to a single processor i860 system. Data partitioning is not a problem but pipelining, etc., is. Targetting F-code to a distributed memory machine is a source of future work.

If a PSP is to use interactive approach, since a PSP is an intermediate language, a PSP program may be significantly different from the high-level language it was generated from. It would not be possible to distribute a program interactively without some means of back-referencing to the original program. A PSP has high-level semantics, meaning that a mapping back to the original program is possible. Otherwise, the PSP must include some referencing information to an original program.

If F-code were to support a high-level language which included directives, they would not be supported. For F-code, and PSPs in general, data-partitioning and parallelization must be automatic or interactive, but not based on directives. Congruence of languages with machines is by automatic parallelization, or interactive parallelization. It would be utopian if this could all be handled automatically.
In order for a PSP to be architecture-neutral, it should reflect at the model level, an estimate of the cost of the underlying execution: congruence must be appraisable in order to choose one optimization, or choice of implementation, over another while targeting the intermediate language to the machine in order to produce the most congruent, most efficient, solution.

**Architecture specific/architecture independent programming environments**

There are a number of low-level machine independent models, the best known of which are the Parallel Machine Interface (PMI) [DW89] and Pi [Wil90], a refinement and extension of PMI. These hardware abstractions describe abstract machines. In the case of PMI, it is a simple message-passing machine. Pi defines ways to emulate a wide variety of operations that manage storage, handle synchronization and communication, manage tasks and capture geometric locality of reference. But it is low level, not a platform for compiler optimizations, merely a hardware-abstraction. PMI and Pi suggest that multithreaded programming models are the best, although the development cost of developing and debugging multithreaded programs are prohibitive. It is important to note that these intermediate languages, because they are low-level, are architecture-specific to loosely coupled message passing machines. They are not, therefore, PSPs.

Linda [CG89] is an abstract programming paradigm gaining widespread acceptance as an effective portable programming environment. It is based on a tuple-pool, and associative matching. Tuples do not have addresses; to match one, one matches field values. Concurrent operation is provided by a process generation mechanism that generates a ‘live tuple’, which is just like any other tuple in the pool, with the exception that some of its components are evaluated by programs, rather than being data values. Much emphasis is placed on compile-time optimization because associative matching is very inefficient. Linda, as a paradigm, is therefore architecture-specific: it is implemented efficiently only a machine which can support associative matching efficiently; if efficient associative matching were a feature of a ‘general purpose’ supercomputer, Linda could be called architecture-neutral. Also, like the expectation of most operating systems, the approach taken is to decompose a problem into tasks, not a decomposition over data; a previous argument gave a reason for PSPs being data-parallel.

A data-parallel intermediate language is used in the prototyping compiler for FORTRAN 90 called YR (Yale intermediate Representation) by Chen and Cowie [CC92]. It is used mainly for simple compiler transformations such as loop interchange [AK84] and loop fusion. YR is rooted in vectorization.

The VSA [Jes90] is an example of an intermediate interface which is low level. VSA describes the semantics of the computational system together with the syntax and semantics of an interface to this abstraction, which allows compilers to generate code for a VSA target system. The interface does not require the definition of an intermediate code, as it provides a set of standard
1.3. PORTABLE SOFTWARE PLATFORMS

code generating routines which must be linked to the high-level language's compiler. It is a platform for imperative languages, a state-transition system, based on user defined, parallel data structures. These represent the state of the system and are manipulated as objects. Updates may be defined over all values comprising an object, or a subset of them, where the subset is defined by an ‘activation set.’ This permits masked assignment. It also permits multithreaded operations: for instance applying arrays of functions to arrays. Geometric operations are low-level and performed by ‘routing’ (it assumes a network), and its opposite, ‘selection’. While it is architecture-independent, it may favour some architectures more than others because it is too low-level. The interface definition is in terms of a stack machine: it was developed with Transputers in mind. Code generation for the VSA is to a stack machine: it produces stack code which is further translated to other forms, like code for register-based machines, data-flow machines, etc.

The most widely known ‘PSP’ is called TDF [TDF91], selected by the OSF for an ‘ANDF’ (Architecture Neutral Distribution Format) which describes a virtual machine with a tuneable architecture which can be efficiently emulated on a wide variety of scalar hardware platforms. TDF is not dedicated to any particular source language; its constructs are suitable for the expression of a range of source languages: C, C++, COBOL, FORTRAN and PASCAL, Ada, SML, SCHEME and LISP. Current work being done with TDF aims to vectorize it [LS92].

TDF constructs are generalizations of the constructs found in different programming languages, designed to satisfy the following requirements:

- All the information that a programming language can represent which helps a code generator produce efficient code should be representable in TDF. This means that programs distributed in TDF can be as efficient as if they were compiled with the best compiler on any target. As such the approach taken by TDF is a low level one, not much different to an architecture-independent assembly language.

- Commonly provided hardware features should be easy to use — for instance the single instruction “array and bound check” provided by many machines.

- As many optimizations as possible should be expressible as TDF to TDF transformations, allowing these optimizations to be written portably. They might be universal (i.e. beneficial for all languages and all target machines) in which case they could be included in a general purpose TDF to TDF optimizer; they might be language specific, in which case they could be included in any of the compiler components for that particular language; or they might be specific to a class of architectures, in which case they could be included in translators for that class of targets.

TDF is low level, and its ‘architecture-independence’ is guaranteed by the ability to generate

---

2It is claimed to be architecture-independent, but this is only true for scalar architectures: it is architecture-specific, dependent on the quality of the vectorizer, since vectorizers are machine-specific
user types (which are called \textit{shapes}) in the most general way. For example, the precision, in terms of number of bits, of floats required for the application is given to the TDF targetter, along with the range of integers possible. TDF does not specify the alignment, since this is architecture-dependent. Sizes of structures depend on the architecture. TDF provides constructs for procedures, pointers, unions, static and dynamic arrays and so on in an ‘architecture-independent’ fashion at a very low level. It is useful for low-level targetters, targetting scalar code to scalar processors, perhaps to vector processors with the aid of vectorization. The problem with TDF is just that: it needs vectorization. It would be ridiculous for a PSP to be scalar if the high-level language it supports is data-parallel. It would be necessary to throw away explicit parallelism to target a high-level language to TDF. Then vectorize it, parallelize it, paralyze it, to execute on parallel machines. It is too low level. Since it relies on vectorization, being incongruent to the high-level language it supports, it is not \textit{architecture-neutral} and therefore not a PSP according to the definition given in this thesis.

Towards \textbf{architecture-neutral} programming environments

MOA [Mul88] is a functional language, based on the data-parallel facilities of APL and defined using a variant of the Bird-Meertens formalism. Unlike APL it features strong typing. There are two main interests with the MOA formalism: it describes arrays in a dimension-independent algebra, and an array operation is defined to have \textit{shape} (called \textit{form} in the literature [HBF92, HM93]) and contents.

There are four main stages to an MOA compiler: parsing, type inference, form inference and code generation.

In [HBF92], it is noted that the usefulness of MOA programming rests on the possibility of statically determined communications. To this end, the compiler has a form inference procedure which predicts the length of lists, the value of forms and the form of arrays (in other words, the shape) whenever possible. Indirect array references’ communication patterns can never, of course, be statically inferred without run-time compilation.

The notion of shape inference is not new: Occam compilers require static memory allocation and some APL interpreters infer forms to minimize space [Bac78].

In MOA an array of any dimension is defined as the tuple $(f, c)$ where $f$ is the form and $c$ is the contents which have a certain lexicographical order. Then $\#f$ is the rank of the array. And selectors can be defined as follows:

$$frm : T^\mathbb{D} \rightarrow \mathcal{F} : (f, c) \mapsto f$$

where $\mathcal{F}$ is the set of $\textit{forms}$. Forms are vectors like $[2, 2]$ meaning a $2 \times 2$ array. $frm$ returns the form of an array, and $cnt$ returns the content of an array:

$$cnt : T^\mathbb{D} \rightarrow T^\omega : (f, c) \mapsto c$$
1.3. PORTABLE SOFTWARE PLATFORMS

\[ \text{siz} \text{ returns the length of the content:} \]

\[ \text{siz} : T^O \rightarrow N : (f, c) \mapsto \prod f \]

It is then possible to define APL operators inductively such as the ordinal list constructor (\(\text{=}\)) where \(\text{=} = [0, 1, 2]\), etc.

\[ \text{=} : N \rightarrow N^* : \text{=} \textbf{=} [] ; \text{=} (n + 1) = \text{=} \textbf{=} [n] \]

where \(\text{=}\) is the list concatenate function;

and the reshape operator of APL (\(\text{r}\)): \(\text{r} \text{sp} : T \rightarrow T^O \rightarrow T^O\):

\[ \text{r} \text{sp} f A = (\prod f = \text{siz} A \rightarrow (f, \text{cnt} A); \Theta) \]

where \((p \rightarrow A; B)\) is an abbreviation for if \(p\) then \(A\) else \(B\). Which leaves the contents of \(A\) unmodified in the result, and gives it a new shape (whose siz must be the same as the original). In this example \(\Theta\) is an empty array which is the result if the rsp cannot be done.

MOA also includes reductions, parallel prefix and indexing (selecting one or more successive layers of an operand) defined in this particular algebra. It is a demonstration of a functional programming language augmented with array features.

One useful facet is the introduction of second-order operators: second-order operators provide language primitives to express communication structure [Ble87, Ski90]. Reduction of a two-dimensional array in a square mesh may be done in two separate steps first column-wise and then row-wise and these two steps are explicitly represented by the algebra using a recursive application of reduce such that reductions are only along one dimension of the array at one time.

There are simplicities in the approach: for example, operations between data-parallel operands assume matching shapes. There is no concept of shape coercion. This is a problem due to the simplistic way in which data-parallel operations are defined: arrays are lists with forms. The elements are always lexicographically ordered. This is why the definition of reshape above is particularly gruesome. A far better treatment of data-parallelism is one where data-parallel functions are homogeneous arrays of scalar functions with indices; Nothing to do with lists at all. This argument will be developed in the next chapter, in section 2.1.1. One problem of MOA's algebra which renders it inapt as an execution model is the restrictive nature of the concatenate operator (as noted in [Sha93]) which concatenates and combines only adjacent elements (having to wait for adjacent elements to be evaluated), rather than an approach based on functions which could evaluate all elements concurrently with the least restrictive trace. A data-parallel object is the re-shaping of a list into an array. And any problem with lists is equally applicable to arrays. This problem is applicable to all algebras derived from the Bird-Meerten formalism. A functional approach to arrays is far better, whereby every elemental operation on an array is a scalar operation supplied with an index like map functions.
An architecture-neutral programming environment: F-code

The subject of this thesis is the implementation of F-code [BMS92]. F-code was defined at the University of Surrey by Muchnick and Shafarenko, building on extensive work done on the data-parallel language EVAL [MS93], in Russia. F-code is similar to the parse tree of EVAL. There already exists an interpreter for a previous version of F-code. The main efficiency gains of a compiler over an interpreter will be by implementing lazy evaluation; by not using a dictionary to keep extents of objects; by producing an assembly-language version of an F-code program which can take advantage of machine features such as pipelining, which some compilers do not; lastly, by not incurring the interpretive overhead.

![Diagram of F-code system](image)

Figure 1.2: General overview of the F-code system

Figure 1.2 shows an overview of the F-code system. Before the start of this thesis, an EVAL to F-code compiler existed, as did a sequential interpreter for F-code. This thesis shows that F-code is compilable and how it may be compiled. F-code was first compiled to scalar C to demonstrate that the entirety of F-code can be compiled; in order to take advantage of machine features, a lower level intermediate graph representation, called T-code, is used for targetting. This intermediate language is particularly suitable for RISC processors. A T-code graph maintains the parallelism of an original F-code program. While C could be reasonably regarded as a portable assembler for virtually any machine, T-code maintains information which would otherwise have to be re-inferrered by an excessively sophisticated C compiler. T-code answers the question: why discard anything (a tractable representation of parallelism, including known dependencies, etc.), and consequently attempt to re-inferr the everything that was discarded. The thesis shows a complete transformation of F-code to i860 assembly language.

F-code is a PSP which supports imperative languages. Thus it has to include imperative assignment. It is not therefore a pure functional programming language. F-code is the first
successful attempt at defining and implementing an architecture-neutral platform; it is yet to be a commercial success. It is not a panacea: It does not handle inhomogeneous data manipulation or any inherent mechanism to cope with irregular problems, but it does contain a very general set of data-parallel operators for regular, scientific problems. It is more architecture-neutral than FORTRAN standards, because data-dependencies are clearly represented using a LISP-like nested list format, and data-parallelism is inherent. The semantics are declarative. An efficient implementation is inferred from an F-code program. F-code is both informally and formally introduced in chapter 2. Inferring a congruent implementation for a single i860 system is shown in chapters 3 to 6.

1.4 Parallel machines

Since a PSP sits between high-level languages and parallel machines, it must equally be able to support a large number of high-level languages and a large number of parallel machines. Parallel machines are generally fixed-size arrays of processors, operating in one of a number of ways:

- Traditional supercomputers are tightly coupled, shared memory vector-multiprocessors with up to 16 or 64 very powerful processors: CRAY, ETA-10 etc. [HJ88, Laz88] These machines are impressive engineering feats with clock periods of less than 4ns. They have multiple segmented functional units, include vector processing and chaining. Parallelism of operations in a vector computer is achieved in several ways:

  1. Performing scalar and vector operations at the same time
  2. Using several vector pipes to different functional units
  3. Using the result stream from one vector register simultaneously (within the same clock period) as the operand to another operation using a different functional unit.

  This is chaining.

CRAYs are generally register machines and load/store oriented, whereas ETA takes operands from memory with the added advantage that the length of vectors is not limited to some maximum length of a register. Vector functional units require some initial setup period, and it must be assured that the length of a vector operation warrants this setup time.

The idea of using vector units to overlap execution is not limited to supercomputers. Modern microprocessors such as the i860 [KM89] and modified SPARC for the CM5 have adopted pipelined functional units; the i860 is does not possess vector units, but rather a segmented 3-stage floating point ALU which may include chaining. In a sense, whether or not a processor in a particular machine is capable of vector computation has no bearing on the nomenclature of its architecture. A MIMD, SAMD machine could equally contain vector processors. In [Ahm91], vectors are the unit of transaction for data-flow machine.
Arcs may contain vector quantities (references to vectors), and tasks are capable of pipeline chaining by executing more than one node in a graph in a pipelined manner.

- VLIW machines are parallel machines operating synchronously with a shared register bank. They are thus congruent with fine-grain parallelism. Interprocessor communication is therefore done in terms of passing data in registers. Percolation scheduling is applied to these machines to compact code into long words of instructions without breaching necessary data-dependencies [Nic85]. Example VLIW machines are iWARP [Bor88], AP120B and AP160.

The i860 is a superscalar architectures [HP92] which is capable of executing one floating point instruction and one integer instruction in parallel per instruction cycle. Code compaction techniques applicable to VLIW machines are also used in i860 compilers.

- SIMD machines operate synchronously: each processor operates the same program in lock-step. Execution is masked: each processor may be omitted from the execution of a particular instruction slot by setting its associated logical mask. The choice operation from EVAL and F-code, $A = \text{MASK} \ ? \ B : C$ means an array of scalar choice operations: $A_i = \text{MASK}_i ? B_i : C_i$, where if $\text{MASK}_i$ is true, $A_i=B_i$, otherwise $A_i=C_i$. This may be executed on a SIMD machine by the following steps:

1. set mask
2. $A = B$
3. invert mask
4. $A = C$

This is equivalent in terms of where statements to

- $A = B$ where $\text{MASK}=$TRUE
- $A = C$ where $\text{MASK}=$FALSE

Intercommunication between processors is usually nearest-neighbour either on a mesh or hypercube, and ordinarily, there is some combining network for reductions. Usually shifts (alignments) are easy to execute. The best points of SIMD machines are also their largest failings. Pure SIMD machines, machines in which every element performs precisely the same operation at a given time, are extremely easy to design and program because there is no asynchrony, only data concurrency. Conversely, because there is no asynchrony, there is no allowance for ‘functional’ concurrency; and the F-code operator, choice, especially when nested, is executed with a low efficiency, due to sequentialization.

Example SIMD machines are the AMT DAP and original Connection Machines (CM1 & CM2) from TMC. AMT added an 8-bit floating point co-processor to every one-bit processor of the DAP, because one bit processors are hardly suited to perform floating point operations. In order to overcome the fixed size of the machine, AMT introduced an
intermediate called VAP [Fla90], a subset of VSA for SIMD machines (VSA was mentioned earlier), which is the intermediate format for the FORTRAN compiler. It hides the physical number of array processors on board a particular DAP; the number of processors is given as a parameter to the VAP translator at translate time (albeit 32 or 64) as well as parameters for run-time profiling. VAP has yet to be updated to provide a sufficient basis for a C compiler [Fla92]. C* [QHJ88] was developed for the Connection Machine [Hil85] and is based upon a data parallel style of programming that maps every data element to a virtual processor [Ano86] and this two-layer approach has found its way into HPF (see previously). C* has also been compiled to hypercube multicomputers:

- MIMD machines are interconnected arrays of potentially independent processors. The memory of MIMD machines is often arranged not as a single memory visible to all processors, but as hierarchies of memory locations, the majority of memory being more expensive to access than that visible to few processors [KDLS86]. Effort must be made to place data in this hierarchical structure in order to maintain locality of reference as much as possible. Some computational algorithms such as image processing algorithms and thermal conductivity algorithms do possess a high degree of referential locality. Where non-local references occur, a program will only run efficiently if data is partitioned and distributed appropriately to the communication pattern of the application, and the machine message passing structure of the machine.

MIMD machines fall into two general categories — shared memory machines such as Cedar, Tera [CS90], and distributed memory machines, such as the Meiko Computing Surface or Supernode composed of Transputers [Inm85]; and iPSC/1, iPSC/2, iPSC/860 [BH92] based on the 80286, 80386 and i860 respectively.

**Distribution** Most parallelizing compilers allow a program to intimate to the compiler regular distributions like CYCLIC, etc. Parallaxis [Bra91] is an example of a language which allows the programmer to specify arbitrary network topologies by means of a functional description. Compilers are not yet advanced enough to automatically infer appropriate distributions and re-distributions to execute efficiently at run-time, and the burden for this complex problem is given to the programmer. Parallelizing compilers are sometimes interactive, requiring the user to guide the compilation, for example SUPERB [ZBG88] which translates FORTRAN 77 into message passing FORTRAN and the MIMDizer [MIM91] which is an interactive, menu-driven system intended to be a programming environment for constructing parallel programs. Such systems are called semi-automatic parallelization systems. Partitioning the data and processor mappings are the responsibility of the programmer while the system automatically controls the parallel execution according to this explicit partitioning and keeps track of data-placement. Automatic parallelization is not yet possible.
- **Hashing** For a single vector processor and a number of memory banks, one might attempt to find an access scheme which evens accesses across the banks [MV84]. Randomized hash functions must be fast because each memory access must undergo a memory translation of its address via this mapping function. Since it is random, it may also randomly be inefficient, accessing the same bank repeatedly. If a seed is chosen only once per program, this inefficiency would be consistent throughout the entire execution of a particular program. However, if the randomization is made to occur on the entrance of loops, the average execution time would be more likely to prevail.

For a distributed machine hashing or random distributions may be used to even out network use. There is a question, therefore, of not having a programmer specify a particular distribution, but rather randomizing everything.

- **Migration** Another approach to improving execution time is to allow migration: repeated access to a variable along a network in a particular direction could allow a run-time system to make its physical location migrate back along that direction. This is only applicable to large loop repetitions.

Migration must be integrated into an existing routing mechanism, like Valiant's [VB81] where routing consists of two phases: each packet is set first to a random intermediate destination and from there on to its final destination, where both phases are deterministic. This scheme evens out network load and thereby decreases prospective latency.

- **Caching** Shared memory multiprocessors use data caches, and this also applicable to distributed memory machines. The major problems with caches is maintaining cache-coherence. This is simple enough for processors with a single bus and a shared memory (some cache units perform bus-snooping which picks up writes of particular data to the shared memory and either invalidates the local copy or modifies the entry.) But 'casual' snooping is not applicable to distributed memory machines since most communications are not broadcasts. Cache coherence schemes such as LIMITless [CKA90] produce directory schemes which keep references to which node has which data. This particular scheme requires only a small hardware-maintained number of directory entries for cached pieces of data, performing traps to extend the directories indefinitely.

- **SAMD** Machines are the middle ground between MIMD machines and SIMD which may:
  
  - Run in lock-step for SIMD operations
  
  - Run completely asynchronously for MIMD operations
Such an architecture is a CM5, where each processor is a modified vector SPARC which can operate in SIMD or MIMD fashion. Another example is Triton/1 in [PWTH92]. Also bulk-synchronous computers [Val89] fall into this category.

1.5 Compilation techniques

1.5.1 Parallelization

Almost all parallelizing compilers generate SPMD style code and employ the owner-computes rule [CK88] to distribute the program's computation. The SPMD model of computation is the data-parallel model.

Parallelization is preferably automatic, however at the moment only semi-automatic parallelization is possible. Implementations of parallelization are from hand-crafted approaches (which are very machine specific), to support of parallelization by libraries (which are architecture dependent), to semi-automatic parallelization: (SUPERB [ZBG88] and HPF [HPF92] require the user to partition data interactively but do abstract from the architecture; SUPERB includes an interactive analysis tool to set up statistics for the programmer) to the utopian automatic parallelizing compiler which has yet to exist (the problem of automatic parallelization and distribution of data is very difficult if not impossible). Parallelizing compilers also include the use of declarative constructs to guide parallelization.

Basic parallelization employs the following stages [ZC92] with reference to FORTRAN parallelization systems. The source program does not include explicit message passing; and it is the job of the parallelization procedure to introduce message passing automatically, according to the data distribution:

1. The front end first processes the source program and subjects it to conventional analyzes such as control-flow analysis, data-flow analysis and dependence analysis; it also normalizes the program. A normalized program is one such that the initial values of loop variables and increment steps are both 1; the body of a do loop does not contain explicit control transfers; subscript expressions are functions of loop variable(s).

2. The normalized program is then split into a host program and a node program. The host program performs global management tasks, while the node program in the SPMD owner-computes paradigm performs the actual computational task. The owner-computes paradigm is one where computations which define the data elements local to a processor are performed by it.

A typical hypercube, such as the Intel iPSC or the NCUBE/ten has a host processor that manages the I/O devices and the collection of node processors [Pal86].

3. The node program is rewritten in two ways: each instruction is masked to conform to the owner-computes paradigm; and communication (message-passing) instructions are inserted
for all non-local accesses.

4. The node program is then optimized: communication and masking are improved: communication statements are moved out of loops and combined to perform aggregate communication where possible; strip-mining [ZC90] can be achieved in many cases by propagating mask information to the loop bounds. The optimization of communication is usually the most important optimization for distributed memory multiprocessing systems.

Ordinarily, FORTRAN standards for data-parallelism include data distribution directives: normally a number of standard distributions such as BLOCK and CYCLIC are included. Also an annotation to specify the size, perhaps structure, of a processor array. Procedures may be called with distributed arguments; also library functions may be called with distributed arguments (library functions may redistribute arguments to utilize an optimal internal communication structure to evaluate a function and subsequently redistribute the result). Both intra- and inter-procedural analyses are used. If multiple distributions apply to a procedure, runtime or node-splitting techniques such as cloning may be required to generate the proper code for the program [HKT91]. A parallelizing compiler generates a call graph from the initial program: as such a procedure may be called from a number of places with a number of different distributions which it must adapt to, perhaps dynamically at run-time. Also FORTRAN standards may include the forall loop in which all iterations may be executed in parallel, the parameters of which are the iteration set and the processor set across which the forall is applied.

Most FORTRAN parallelizing compilers deal in terms of data-distribution only; however FORTRAN D does things somewhat differently: The problem of data decomposition (collectively the alignment and the distribution) can be approached by considering the two levels of parallelism in data-parallel applications [HKT91]. First, how arrays are aligned with respect to one another, both within and across array dimensions. This is the problem mapping induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program given an unlimited number of processors; it is largely independent of any machine consideration. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of data distribution. This is the machine mapping caused by translating the problem onto a machine with finite resources. It depends on the topology, communication mechanisms, size of local memory, and the number of processors in the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

This two level scheme has the following benefit: data-distribution does not subsume alignment. For instance, a distribution statement alone may not be able to specify that one 2-D array be mapped with the transpose of another.

The reason FORTRAN D included both alignment and distribution specifications is that
it was designed to support programming on both SIMD and MIMD machines. It also supports irregular data distributions and dynamic data decomposition (changing the alignment or distribution of a decomposition at any point in the program).

1.5.2 Code compaction

Code compaction is another name for [fine grain] parallelization. Anantha and Long [AL90] describe that the purpose of compaction tools is to take a sequential program and output a parallel representation — perhaps with human assistance. This is like vectorization, except the goals of code compaction are mainly to schedule VLIW architectures.

Since the i860, a superscalar architecture, is used as the target architecture of a demonstration implementation of F-code, the code generator needs to compact code to be congruent (making use of the ability of the i860 to execute both an integer, and a floating point operation per clock cycle, concurrently.)

With a PSP one starts with a parallel representation and compaction, in a PSP sense, is scheduling, where parallelism and dependencies are already known (and need not be reconstructed). The code compaction techniques described assume a scalar program which is to be parallelized onto a VLIW (superscalar) machine; it needs to reconstruct dependencies from a scalar program. It is shown for comparison.

In code compaction techniques, one starts with basic blocks and Nicolau nodes [Nic85, Aik88, AN88] representing the flow graph of a program to be compacted. In these cases a basic block is defined to be: a group of instructions treated as a unit. It contains at most one branch which is the last instruction. Only the first instruction of a basic block may be the destination of any branch. A Nicolau node is similar to a basic block, except multiple branches are allowed at the end and all statements must be executed in parallel.

The conditional branch is represented by a DAG with n branches and at most n + 1 continuations. In the case where there are no conditional branches, there is a default continuation. The process of parallelization is the process of transforming a graph constructed purely in terms of basic blocks into one constructed purely of Nicolau nodes.

In [AL90] it is shown that this is wasteful of effort — one type of graph should be used instead. They produce, for each line of assembly language, "write-sets", "read-sets" and "dependency-sets". These are lists of line-numbers maintaining dependencies; these are three classes of data-dependency — read and write-sets denote data dependencies and data anti-dependencies. Since this approach also relies on a text-based symbolic intermediate (assembly) language it gains unnecessary 'false' dependencies from symbol-name clashes. Dependency-sets are constructed for the purposes of percolation [Nic85, AN88] — only backward dependencies are used for compaction:

\[
\begin{align*}
0 & T1 = 800 \\
1 & T0 = 1 \\
2 & T1 = T1 + 1 \quad \{0\}
\end{align*}
\]
Obviously, by giving only backward dependencies, scheduling requires percolating instructions which can be executed concurrently backwards as far as the last back dependency. This can result in the VLIW schedule:

\[
\begin{align*}
0 & \quad T_1 = 800; \quad T_0 = 1 \\
1 & \quad T_1 = T_1 + 1; \quad T_0 = T_0 + 5 \\
2 & \quad T_0 = T_0 + 7 \\
3 & \quad \text{EXIT}
\end{align*}
\]

After the construction of these dependence sets unimportant lines of code are removed (those with no write-dependence) — this is called dead code elimination. To overcome symbol-name clashes variable renaming is introduced. Code compaction is quite a mechanical operation comprising of the application of a small set of graph operations.

### 1.5.3 Loop parallelization

Fine-grain (instruction-level) parallelization (compaction) captures irregular parallelism inside a loop body, however it is not effective across loop iterations. In an attempt to bridge the gap between fine- and coarse-grain loop parallelization Aiken and Nicolau [AN88] present a scheme for producing a time optimal execution schedule given a loop and a set of dependences between its statements. It is mainly intended for synchronous parallel machines.

The scheduling restrictions of loop parallelization are loop-carried Dependencies (data-dependencies between loop iterations) [AK84]. The normal way of approaching the problem of loop parallelization is to execute the iterations of a loop on several processors, subject to the constraint that loop-carried dependencies are not violated. It is also important to note that for pipelining purposes (even on a single processor) loop carried pipe-line dependencies are equally important for efficiency: long, well-pipelined loops are better than short loops causing pipeline bubbles at the end of iterations.

To make the loop parallelization task easier, one assumes that loop-carried dependencies are only from one iteration to the next; Munshi and Simons [MS87] have observed that loops encountered in practice can be converted to this form by loop-unrolling.

Parallelization proceeds by creating a partial execution history of a loop, say the first \( i \) iterations, and then by scheduling the statements of those \( i \) iterations as early as possible in a greedy schedule: If the longest chain of dependences on which a statement \( x \) depends has length \( j \), then \( x \) is scheduled at time \( j \). Occurrences of an individual statement exhibit a pattern once a sufficient number of iterations have been scheduled.
1.6 The aims of this thesis

There are a number of aims of this thesis. This introductory chapter has fulfilled one of those aims: to expatiate the idea of architecture-neutrality. F-code is defined in the next chapter. The next aim of this thesis is to show that F-code is an architecture-neutral platform. This is done in two ways: (1) by implementing it on one architecture, and discussing its implementation on a further variety of machines, to prove its architecture-independence (2) by showing the abstract way in which F-code is implemented to show its architecture-neutrality.

It is the particular aim of this thesis to show how F-code can be compiled to a single RISC processor, which is a sequential, pipelined, superscalar, implementation; architecture-independence is shown by discussing the implementation of F-code on a distributed machine, which is 'antipodean' to a sequential implementation.

It will be shown that the implementation of F-code infers scalar programs by a procedure not unlike parallelization which are then sequentialized for a sequential machine or executed in parallel on a parallel machine. Therefore, the procedure for implementing F-code on a sequential machine are not dissimilar to that for a parallel machine.

In summary, it is the aim of this thesis: (1) to demonstrate that F-code can be compiled congruently to a RISC processor (2) to show F-code's architecture-neutrality, thereby to demonstrate the fact that it is a PSP.

The RISC processor used in this thesis is the i860; however, the implementation is more general than that. The code generator produces an abstract machine graph which subsumes the functionality of all RISC processors, which is moulded to the particular processor type at the back end of the compilation procedure. Architecture-specific and processor-specific aspects of the compilation process are withheld to the back end of the compiler. Most of this thesis, therefore, is involved with compilation in the abstract.
Chapter 2

Introduction to F-code

- 2.1 — Introduction to F-code ......................................................... 29
- 2.2 — A quick guide to implementing F-code ............................................. 53

2.1 Introduction to F-code

F-code is a portable-software-platform for data-parallel languages. The last chapter described what a PSP must possess to be called a PSP, and this is not discussed again here; most importantly, F-code is a PSP for imperative data-parallel programming languages, and hence it is a notation for describing computation on data-parallel objects and it includes explicit (data-parallel) assignment. Since it is a PSP for imperative programming languages, it does not, when viewed as a whole, possess referential transparency. If this lack of referential transparency is not considered thoroughly at the outset it can easily hinder the usefulness of a PSP as a basis for reflexive optimization. One of the main uses of a data-parallel PSP is for architecture-independent data-parallel optimization. The BNF syntax for F-code is given in appendix A.

The intention of a data-parallel PSP is identical to those of current FORTRAN standards: to foster a machine-independent environment in the domain of parallel computers. There is a far more general set of operations in F-code than in any high-level language. Most importantly these are rigourously defined in terms of data-parallel algebra, giving any F-code program an unequivocal functional specification. The data-parallel algebra is defined in appendix A. This algebra is, in a later chapter, shown to be a useful tool for optimization, both to represent an optimization and to prove its efficacy. F-code is also an algebra in its own right: transformations can be applied to F-programs for optimization or targetting purposes.

There are two kinds of data: the array and the record. There is no provision for arrays of arrays, other than by forming arrays of records. Functions are not first class citizens; there is no way to have arrays of functions, or use functions as arguments, or return functions as the result of functions. A data-parallel object (an array) is an object which is nonscalar or, to make it regular, scalar. F-code does not restrict the rank (the number of dimensions) of an object all. It is just as easy to manipulate a scalar (with rank zero) as it is to manipulate a vector (with rank
one), or an object of rank two (a matrix), or three (a box), or ten, or even one hundred. It is also
a dynamic language: the shape of an object (an array of its extents) is permissibly known only
at run-time. When records are accessed, the result is always a homogeneous array. Arithmetic
and geometric functions are thereby always on homogeneous arrays. The F-function responsible
for accessing records is select, defined on page 218. Functions are never components of arrays
or records.

The semantics of F-code are wholly compatible with LISP, and it can be considered to be
a kind of meta-LISP. Like [ORH93], it includes assignment into a functional language setting.
Spatially (across data-parallel assignments), it is evaluated lazily: only those parts of the com-
putation which have a bearing on the result are evaluated. This 'selective evaluation' is inferred
at compile time.

2.1.1 Atomicity

F-code is a PSP for data-parallel programming. It is a target for compilers for either scalar lan-
guages via vectorization (a vectorizer is external to an F-code compiler), or more naturally for
data-parallel programming languages. Data parallelism is expressed by operations on nonscalar
objects: Adding two matrices is not an explicit sequential loop in F-code, but rather an atomic
action performed on all matrix elements simultaneously. This is what it does at least conceptu-
ally; its implementation may be entirely different. Since F-functions are conceptually parallel, it
implies that F-code does not impose any ordering on the element-wise operations. (This is un-
like the more restrictive concatenate-based data-parallelism of Bird-Meerten formalisms.) This
concurrency may be thought of as splitting the thread of control for each nonscalar operation
and pulling it together when the operation has been completed.

(C equivalent) (F-code program)

```plaintext
for (i=0;i<....;i++) {
  for (j=0;j<....;j++) {
    ...=C[i,j] + (A[i,j]*B[i,j]);
  }
}
```

(F-code program)

```plaintext
(dyadic add
  (var value C)
  (dyadic mul
    (var value A)
    (var value B)
  ))
)
```

Only the functional specification of the F-code program is given, and only this must be
complied to in a coherent way. There are, of course, very many ways of implementing this
specification. In the sequential case there are two (since there are two mathematical operators
in the tree) classes of implementation. The first is as above. The second is, for a simple vector
implementation:
2.1. INTRODUCTION TO F-CODE

Both this and the previous implementation are equally valid, and fulfill the functional specification. The for loops are for a sequentialized implementation on a scalar processor. Equally, these may be dimensions for data partitioning and distribution across a distributed machine, or dimensions taken up by vector operations. It is the job of an F-code compiler to infer the extents (shapes) of operations (which are shown as \( \ldots \) in the C equivalent programs).

2.1.2 Architecture neutrality

The last section, 2.1.1, gave a first glimpse of how architecture-neutrality is attained in F-code. The algebra has obvious descriptive-simplicity, since it treats data-parallel objects without reference to indices. Architecture-independence is achieved because the algebra can be implemented in any number of ways — in any number of architecture-specific ways. Architecture-neutrality is achieved because the algebra may represent data-parallelism in a language-neutral way; data-parallelism is the commonality between all imperative high-level languages.

2.1.3 Functional parallelism

F-code only contains necessary (object-wide) dependency information. Because F-code is represented in list form, the dependencies are explicit and hence there is explicit functional parallelism. A small demonstration of functional parallelism is given below. The LISP expression:

\[
(TI\text{M}\text{ES} \ (DI\text{FFER\text{E\text{NCE}} \ A \ B}) \ (ADD \ C \ D))
\]

can first compute the difference and then the sum, or the other way around. The same may be true with an imperative language. The FORTRAN expression:

\[
(A+B+SIN(X**2))*(G/A*EXP(-R))
\]

may evaluate the first bracket first, or the last. The result should be no different; however, with finite precision arithmetic, rounding errors may depend upon ordering: \( A(B-C) \) may be nothing like \( (A+B) - (A+C) \) for example, if \( A \) is big and \( B \) and \( C \) are almost equal and small. In an imperative language, the ordering is important since imperative languages allow the use of side-effecting functions, which is why this concurrency is called functional. Pure functional languages possess referential-transparency [PJ88] — they do not allow side effects.

For vector processors, functional concurrency can be used for pipeline chaining: the solution is to use a few pipes feeding the leaves of the arithmetic tree to one end and multiplexing the
CHAPTER 2. INTRODUCTION TO F-CODE

other end not to memory but to a different pipe. This sometimes shows an impressive 10-fold reduction in pipeline starvation time [HJ88].

Functional concurrency is helpful for array processing on processor networks since it can be exploited to hide latency [BMS92].

Functional concurrency is best denoted using a list format, similar to that of LISP. Each vertex of the tree is a function (F-instruction) invocation with its subtrees executing in parallel for most of the vertex labels (or F-instructions). This way, the syntax provides an adequate framework for expressing functional parallelism.

2.1.4 Rigid type hierarchy

![Type Compatibility Diagram]

If a language is to be well defined, the compatibility between its data-types must be decided upon. F-code types are subdivided into a fixed set of "strong", "basic" types which form a compatibility hierarchy:

logical < character < integer < real < complex

There is also ample provision for structured data. See section A.1 and [BMS92] for more information.

F-code only includes necessary type-information and one stage of compiling F-code is a type inference stage. The type-information which is and has to be explicit in F-code is that which cannot otherwise be inferred, and nothing more. This is to keep the representation as succinct as possible. A value lower down the hierarchy may be coerced to a higher one in any operation:

(dyadic add
 (var value A)
 (var value B)
)

Suppose that it can be inferred that the type of A is integer, and the type of B is real, then A must be coerced to real before a real addition can take place. An integer addition would not be
suitable because $B$ would need to be coerced to \textit{integer} before the addition. The hierarchy thus provides a mechanism for coping with such problems properly. An addition in this example must have its operands \textit{coerced} to the highest common type between its operands, and the addition must take place using this type and the result is of this type.

The \texttt{var} function, which takes a reference to an object in F-code may have other keywords (corresponding to language context positions) than \texttt{value}, in which case the type-hierarchy is reversed. Section 2.1.6 deals with this.

The type hierarchy provides, along with a strict definition of the language, a natural way of inferring all types in the F-tree to make it purely monomorphic and hence compilable with a monomorphic (hence faster) implementation. Of course it would be possible to provide mechanisms for managing polymorphic instances of type, but that is not the aim of F-code since it is a PSP for strongly-typed imperative programming languages.

### 2.1.5 Geometric operations

It is quite common for a parallel programming to include only rudimentary extrinsic functions for geometric operations, but F-code is quite different. It is the intention of F-code to provide the fullest, most general set of geometric operations possible. Secondly, the left hand side of assignments are also (this is described in the next section) considered to be objects upon which geometric operations can take place. This is quite irregular for imperative programming languages, but as will be seen in later chapters, it can be implemented efficiently and is a source of extra parallelism. Not only that, but it improves language-neutrality, by making the programming language more congruent with the programmer's ideas.

The simplest types of geometric operation are those which limit an object in some way: for example, the F-function \texttt{sect}, which takes a layer of an operand and it is demonstrated in figure 2.2.

The three parts of the figure, show \texttt{sect} applied to a vector, to a matrix and to a box respectively (which are \texttt{EXPR.a} in the expressions). The first parameter of the \texttt{sect} is the dimension to which the \texttt{sect} applies, the second is the F-function which returns the object to
which the sect is applied and the third is an index into the object along the specified dimension. This function, sect can then obviously give any layer of its operand; the rank of its result is one less than the rank of its operand.

A second example of this sort is the geometric operation, called diag which takes a diagonal of its operand. It is shown for both the matrix and box versions. The result of these is always a vector (the shaded area). More variations of diag are shown in a later section when orientation is considered, which can produce objects which are not vectors.

The third part of the figure is the combination of first a sect and then a diag, which shows that these operations can be combined. This operation again reduces to a vector. Remembering that the F-code program is just a functional specification, the grey shaded area is the active computation area, and the geometric operation must only select this area of its object in order to evaluate the F-program.

The F-function comp composes its operands together to produce a larger operand. Figure 2.4 shows all five different way of composing two $3 \times 3$ matrices. All these — in fact all possible variations of compose — are describable succinctly in F-code. It is useful to remember that all geometric operations are nothing more than a definition of address arithmetic, which may be explicitly executed or not, for right and left hand sides of assignments. Alternatively, they may
be viewed as a way of forming apparently complex iteration spaces. It is just as possible to compose two left-hand sides together as it is to compose together two right-hand sides.

If there are, for instance, two independent (data-parallel) assignments, on two vectors. \( L_1 \) is assumed to be the same length as \( R_1 \), and \( L_2 \) is assumed to be the same length as \( R_2 \).

\[
L_1 := R_1; \quad L_2 := R_2 \quad \text{could be written} \quad L_1, L_2 := R_1, R_2
\]

It is preferable to represent this in F-code using two composes, and it looks like:

\[
\text{assign} \\
( \text{comp 0} \quad L_1 \quad L_2 ) \quad \text{-- L1 --} \quad \text{-- L2 --} \\
( \text{comp 0} \quad R_1 \quad R_2 ) \quad \text{-- R1 --} \quad \text{-- R2 --} 
\]

The syntax of F-code is not important at this early stage of the chapter, however it may be quite obvious. Basically, \( L_1 \) and \( L_2 \) are composed and form the left hand side of a data-parallel assignment and \( R_1 \) and \( R_2 \) are composed and form the right hand side. The shape (the iteration space) of the assignment is easily inferred from the shapes of \( L_1, L_2, R_1 \) and \( R_2 \). The length of the assignment is the extent of \( L_1 \) plus the extent of \( L_2 \), which is identical to the extent of \( R_1 \) plus the extent of \( R_2 \). Needless to say, all of these \((L_1,L_2,R_1,R_2)\) are arbitrary F-code expressions which may be geometric expressions for the left-hand side or a mixture of geometric and computational expressions for the right.

There are several more, general data-parallel geometric operations. As a final example, the slice (page 223) operation is demonstrated:

![Slice diagram](figure2_5)

**Figure 2.5: The F-code operator slice**

slice takes three operands: \((\text{sect d} \quad \text{EXPR}.a \quad \text{EXPR}.i)\) — see figure 2.5 for an example. The first is the dimension of the slice. The second is the object to slice \((\text{EXPR}.a)\), and the third is a vector \((\text{EXPR}.i)\). \text{EXPR}.i is used to choose layers of \text{EXPR}.a. It is very easy to see that a slice is identical to a number of sects and a number of composes, however function slice is something far more succinct and tractable. F-code does not aim to provide the minimum set of geometric operators possible, but rather a more natural approach: a notation which uses
recognizable high-level data-parallel operators. Another reason for keeping high-level operators explicit is that slice could be a basic operation of the machine and hence it is far more useful to keep data-parallel operators explicit instead of performing a complicated, hence unsatisfactory, pattern matching algorithm on the tree to elicit them. Of course, this does not mean that F-code includes an architecture-specific function; it can be congruently implemented on any general-purpose machine.

2.1.6 Sort

Data-parallel computation normally involves geometric transformations that select and reorder elements of nonscalar objects. The results of such selects can act as objects in their own right. It is important to provide some sort of address arithmetic. Since both the left hand side and the right hand side of assignments can be geometric operations (section 2.1.5), one needs the notion of sort. There are three types of sort in F-code, viz. value, name and target.

Each of these corresponds to a different context position in assignments:

value value is the most obvious of the sorts, and corresponds exactly to the right-hand-side of assignments. Another name, in other literature, for it is rvalue. Quite simply, it is what one would normally call a data-object. value objects can be operated on either by computational (like addition) or geometric (like compose) operators.

target target corresponds to the left-hand-side of assignments. Most languages do not permit left-hand-sides of assignments to be anything but variable names. F-code is different and allows the left-hand-side to be any expression containing only geometric operations (like compose). A target is basically an address (another name for which is lvalue), and this may take the value hole which is a dummy reference.

name name is similar to a target, except it can additionally be dereferenced, and yield a value. It combines the functionality of target and value. It is described later.

The type-hierarchy for target and name is opposite to the one for value, such that:

\[
\text{logical} \succ \text{character} \succ \text{integer} \succ \text{real} \succ \text{complex}
\]

The reason for this reversal is shown by considering the simple assignment \( L := R \). If \( R \) is of a lower type than \( L \), it could be coerced upwards, so that they are of the same type, and the assignment is atomically of this higher type. If \( R \) evaluates to an integer type, yet \( L \), the left hand side of the assignment is a complex type, all elements of \( R \) are coerced to complex before the assignment. This is perfectly correct due to the hierarchy.

Alternatively, if the type of \( L \) is again higher than the type of \( R \), \( L \) could be coerced downwards (thus with the opposite type-hierarchy), and the assignment be in terms of the lower type. Thus, if \( L \) can legitimately be coerced downwards this is done instead. This solution usually requires
2.1. INTRODUCTION TO F-CODE

a smaller amount of memory, because the size of higher types of course generally require more memory.

One may ask why only five different types are available in F-code, and realize that they are similar (except for complex) to the FORTRAN and C types. There is no attempt to provide precision mechanisms for floating point (real) numbers. Merely, there is a uniform type called real, whose precision is fixed (not to say that it cannot be single, or double, or any precision in actuality). There are two types of integer provided: the standard types character and integer, and the Boolean type logical. The lengths of these are assumed to be architecturally dependent, or rather the way they are implemented is. character is generally taken to be a byte, and integer is generally taken to be a word, and if it is possible logical is taken to be a bit. But talking about bit-lengths is purely a digression.

2.1.7 Context positions

There are three context positions in an assignment, and these are modelled in F-code by the three sorts. In the general data-parallel assignment (1. in the figure), the right hand side always

\[ A := B \cdot C \]

1. Data-Parallel assignment

\[ A := \text{SAME} \cdot C \]

2. Optimized

Data-parallel assignment
( Cases where SAME is on the RHS. )

Figure 2.6: Context positions

has to yield a value which is assigned to memory via the left which always has to yield an address. The right-hand side must be composed of values, or names dereferenced. If a right-operand is originally a value it is said to be in the hard context position; the left-hand side is always an address and is said to be in the soft context position; finally, if part of the right hand side needs to be dereferenced (if it is a name), it is said to be in the double context position.

Names can be dereferenced, and must for this reason not include dummy references. When one creates an object in F-code, this creates both a value object and a name object which refers element-wise to the values. This is only a conceptual description: it can be implemented very differently. Thus, when one dereferences an object using a name, it "looks-up" elements through the name. The target on the left-hand side is just an address (or data-parallel object, including data-parallel geometric operations which yields addresses).

The second (2. in figure 2.6) example is different. The SAME on the right-hand-side means that this right-hand object is the current value of left-hand-side. In this case, for optimization
purposes (i.e.— one does not want to repeat all the address arithmetic for the right-hand SAME),
the left hand side is *double*. The *double* is both a *value* and a *name*, the name is relevant for the
left-hand side and the dereference of it is appropriate for the right-hand side.

The fundamental difference between *targets* and *names* which may seem at first glance to
be very similar is their context positions. *targets* are always on the left-hand-side of assignment
whereas *names* can be used similarly on the left-hand-side, or dereferenced and used on the
right-hand-side. There is further distinction, which is this: *names*, as said previously, can be
dereferenced, and their dereferenced types and their *name* type must be same. The coercion
mechanism (for a function like compose) which can compose *names* together, asserts that the
two *name* objects are of the same type. So that they can be dereferenced unambiguously, and
will subsequently be of the same type as the *names* — coercion on values chooses the highest
type, coercion on names chooses the lowest type, and there could possibly be an incongruity
here. Function compose works differently for *targets*: when *targets* are composed, they retain
their original types (thus the function composes an operand which in inhomogeneous in term
of type in only the scope of the left-hand-side of an assignment). The right-hand-side of an
assignment, since it is a value is always coerced to the maximum type.

![Figure 2.7: A general data-parallel assignment](image)

Figure 2.7 above, shows a general data-parallel assignment. The *value* is written to memory
via the *target*, which includes for the example dummy references. In fact, it looks quite similar to
masked assignment, however, it far more general purpose. Since, a *target* is being used, arbitrary
elements of it can be of any type. The right-hand-side is coerced to be, or is naturally, uniformly
*integer*, however parts of the *target* are *integer* and parts are *real*. Those elements which are *real*
require an elementary type coercion between *integer* and *real*.

If one considers what would happen if the *target* is dereferenced (mentally reverse the arrows),
it is feasible, and the type of the dereference is *real*. There is thus a dichotomy between the read
(dereference) type, and the write type. This need not necessarily cause a problem at all, except
when procedures are used: *(since F-code is derived from some of the ideas of EVAL, a simple*
2.1. INTRODUCTION TO F-CODE

A piece of EVAL is used to demonstrate this particular nodus.

PROCEDURE P(<> X)
X : real

This declares an EVAL function P, and declares that X's read and write type is real. Thus, one would like to dereference X and it is has previously been shown that it is possible to have different read and write types if one allows dereferencing of targets. Clearly this is possible, but not nice. Effectually, X is in a double context position, and it should instead be represented by a name and not a target. Recapping, the difference between a name and a target is that names can be dereferenced, and are of a uniform type. One cannot compose a name of different types, unlike the situation shown above in the figure — and hence the read and write types are always identical. Thus, the distinction (useful as it is) between names and targets is not a conceptually difficult one, and the two similar notions exist for optimization purposes, and as just described for procedures.

2.1.8 Rank coercion and operation orientation

It is useful to define a uniform procedure for defining/inferring the rank and the shape of the results of F-expressions. This section deals with inferring the rank of an object. Section 2.1.9 deals with inferring the shape. Operand orientation is a means of providing a uniform approach to rank coercion, as will be shown in this section.

The example below shows the addition of two F-expressions which return matrices. In the example, implicit indices are arbitrarily labelled i, j, k, etc. A and B are arbitrary F-expressions. F-code provides a transpose function (formally defined on page 222) which transposes an object. There are a number of ways of adding two matrices together, depending upon how the operands are orientated. The example shows the algebraic qualities of F-code:

(dyadic add
  (A) A[i,j]+B[i,j]
  (B)
)
(dyadic add
  (A)
  (transp 0 (B)) A[i,j]+B[j,i]
)
(dyadic add
  (transp 0 (A)) A[j,i]+B[i,j]
  (B)
)
(dyadic add
  (transp 0 (A)) A[j,i]+B[j,i]
  (transp 0 (B))
)
The transpose does a transpose operation, and thus conceptually does nothing more than re-order the indices. The last of the four cases could be written as the identical functional specification, which may or may not be identical implementationally:

```
(transp 0
  (dyadic add
   (A) A[j,i]+B[j,i]
   (B)
  )
)
```

F-code introduces the most general possible mechanism for rank coercion. It is always useful to be able to apply operations on objects of different ranks; for example an operation between a matrix and a vector. The following example adds a matrix A to a vector B, resulting in a matrix.

```
(dyadic add
  (A) A[i,j]+B[i]
  (B) A[i,j]+B[j]
)
```

The masks 01, 10, or any binary number provide a mask to the sets of indices. In the first of the above examples, the mask is 10, and the i index is selected to index B. In the second of the above examples, the mask is 01, and j index is selected to index B. The mask simply denotes active dimensions. The vector B can be considered to be applied along a particular dimension of the matrix A. This is why it is called orientation. The combination of transp, and orientation masks provide for any possible orientation.

Normally, a masked expression, such as "010 (X)" has a rank which is the length of the mask (3 — hence a box), while the F-expression (X) has a rank which is the number of ones in the mask (1 — hence a vector). This is part of the means for inferring the rank of the result.

```
(dyadic add
  011 (A) A[j,k]+B[i,j]
  110 (B)
)
```

```
(dyadic add
  101 (A) A[i,k]+B[j,k]
  011 (B)
)
```

The masks 01, 10, or any binary number provide a mask to the sets of indices. In the first of the above examples, the mask is 10, and the i index is selected to index B. In the second of the above examples, the mask is 01, and j index is selected to index B. The mask simply denotes active dimensions. The vector B can be considered to be applied along a particular dimension of the matrix A. This is why it is called orientation. The combination of transp, and orientation masks provide for any possible orientation.

```
(dyadic add
  011 (A) A[j,k]+B[i,j]
  110 (B)
)
```

```
(dyadic add
  101 (A) A[i,k]+B[j,k]
  011 (B)
)
```

The examples above, then each add two matrices to produce a box with the particular orientation denoted.

Orientation masks are always used to orientate, thereby increase the rank of an operand. In order to reduce the rank, there are a number of mechanisms in F-code, the simplest of which is the sect function, previously introduced. In the following example, the F-expression, A, is a box; the example returns a matrix (it reduces the rank by 1):
2.1. INTRODUCTION TO F-CODE

\[
\text{sect 1} \\
\quad (A) \\
\quad (\text{const 2}) \\
\]

The definition of \text{sect} fixes dimension 1 of \( A \) at the value 2. Dimensions in F-code are numbered from 0. The result of the operation is a matrix, which uses the indices \( i, j \). \( A \) is then indexed with \( A[i,2,j] \). This is given a formal treatment in appendix A, on page 213.

There are some other functions which use masks for orientation purposes whose rules are different. For example, if one considers what happens for the geometric function \text{diag} (figure 2.8):

![Diagram](image)

Figure 2.8: Orientation for \text{diag}

The orientation masks in these cases define which dimension is to remain intact (in which case, the mask-bit is a 0), and which dimensions are to take part in the diagonal. The default mask for the box example, above would be "111", and the \text{diag} would select a vector diagonal from the box, whereas those in the example select a matrix (effectively an array of vectors, where the vectors are evaluated from the diagonal).

Finally, consider the function \text{reduce} which does a mathematical reduce. Figure 2.9 shows, first of all, a couple of the ways of implementing a \text{reduce} on a vector to produce a scalar result. The bottom three parts of the diagram demonstrate a \text{reduce} on a matrix. The default case is the the reduce works along both dimensions (since there is no mask), and results in a scalar. Finally, where \text{reduce} has an orientation mask, a 0 in the mask denotes a dimension which does not take part in the compose, and obviously a 1 in the mask denotes an active dimension of the reduce. Masks in the \text{reduce} and \text{diag} denote active dimensions, not really 'orientation'.

2.1.9 Shape coercion and lazy evaluation

If a data-parallel operation is executed with operands of different sizes, there is a rule defining the size of the result. Thus, as well as a general mechanism for rank coercion, a method of shape 'coercion' is also needed. For dyadic operations, the result should only be computed where the 'iteration-spaces' of its operands overlap, thus avoiding wasted computation (and
There are several ways of implementing reductions.

(REDUCE EXPR.a) (REDUCE EXPR.a)

(REDUCE EXPR.a) (REDUCE 10 EXPR.a) (REDUCE 01 EXPR.a)

Figure 2.9: The reduce function

communication). This is part of lazy evaluation. The shape of an object is an array of its extents, such that a $10 \times 20$ object has the shape $[10,20]$.

(dyadic add
   (A)
   (B))

If A were a $[3,5]$ object and B were a $[6,4]$ object, the size of the result will be $[3,4]$ since $\min(3,6) = 3$ and $\min(5,4) = 4$. This is shown in figure 2.10.

One of the key concepts of F-code is the ability to compute data-parallel operations in a lazy manner. This does not mean to say that operands are computed when they are needed, rather they are computed eagerly, but only for relevant parts of the iteration space.

In the above example, though the F-expression B returns a result of size $[6,4]$, only the shaded area (half of it) needs to be computed in the lazy case: an object of size $[3,4]$. This dramatically cuts down the amount of active computation needed to compute the result, and only those which make any difference to the result are computed.

Rank coercion is also important to lazy evaluation. In an example with sect:

(sect 1
   (A)
   (const 2) )
2.1. INTRODUCTION TO F-CODE

If one assumes \( A \) returns an object which is box, and the \( \text{sect} \) therefore returns an object which is a matrix. In fact, due to lazy evaluation only that layer of \( A \) which is relevant to the result needs to be computed.

If \( A \) is a box of shape \([s1, s2, s3]\), the \( \text{sect} \) returns a result which has a shape \([s1, s3]\), since a \( \text{sect} \) is being taken along the first dimension (dimensions are numbered from zero). Only \( \frac{1}{12} \) of the result of a wasteful non-lazy implementation needs to be computed.

2.1.10 Ordering

F-code is not a purely functional language, and thus includes the possibility for side-effects with particular F-functions. Before considering assignment, there are a few F-functions which introduce ordering (or lack of ordering) within F-trees explicitly: namely \( \text{seq} \), \( \text{par} \), and \( \text{comma} \). Arithmetic and geometric operations do not enforce an ordering between their operands: they exhibit functional concurrency. They are evaluated from the leaves upwards, but not in any specific order between branches.

\( \text{seq} \) sequentializes its operands, which are F-functions that return scalar values (see page 229 for the meaning of these values). Logically, the only purpose of these F-functions is to carry out assignments. This is not a restriction of the semantics of F-code, but a logical one.

\[
(\text{seq} \\
(A) \\
(B) \\
(C))
\]

The program above performs \( A, B, C \) in that order. The result of \( \text{seq} \) is a scalar number (see page 229.)

\( \text{par} \) performs its operands in any order (see page 229.) Logically, again, the only purpose of these F-functions is to carry out assignments.

\[
(\text{par} \\
(A) \\
(B))
\]
This program possibly performs A and B in parallel. A and B are F-trees which carry out assignment. `par` may introduce non-determinism if the targets of assignment A and assignment B happen to coincide. The F-code compiler does not reject such programs, but their results may be non-deterministic. The result of a `par` is a scalar value (see page 229.)

The last F-function specifically for ordering is `comma` (see page 229.) `Comma` sequentializes its arguments, returning the data-parallel value of one of them, discarding the other.

```plaintext
(comma left (A) (B))
```

Computes A, then computes B, then returns the value of A, the left argument, which is an array. The value of B is discarded. Logically, again, B is only going to be an assignment. Effectually, this inserts a side effect after evaluating A.

```plaintext
(comma right (A) (B))
```

Computes A, then computes B, then returns the value of B, the right argument, which is an array. The value of A is discarded. A is only going to be an assignment. Effectually, this inserts a side effect before evaluating B.

`Comma` is used to insert a side-effect in any data-parallel geometric or arithmetic F-tree, at any point. They are more likely to occur towards the leaves of an F-tree, but the semantics of F-code do not exclude them from appearing anywhere.

By way of example:

```plaintext
(dyadic add
  (comma left
    (dyadic add
      (A) (B)
    )
    (C)
  )
  (comma right
    (D)
    (dyadic add
      (E) (F)
    )
  )
)
```

C and D are logically assignments, but the semantics do not exclude them being any F-tree: the current compiler, however, restricts them to returning scalar values (as does assignment).

The ordering of this tree is `((A[|B];C)][|D;|E|F))`, where `|` represents [possible] parallel execution, and `;` represents sequentialization. The program may be non-deterministic if the targets of assignment C, and assignment D overlap. The duty of a front-end compiler which produces F-code is to ensure that such targets do not overlap.
2.1.11 Assignment

The assign function of F-code introduces data-parallel assignment. (See page 228.) The left and right hand sides of assignments may be data-parallel objects which undergo shape coercion like any other application of one operand to another.

\[
\text{(assign} \quad (A) \quad (B) \text{)}
\]

Assigns the value of B to the target evaluated by A. A may include geometric F-functions (such as sect); B may include both geometric and arithmetic F-functions. Lazy evaluation is implemented for both the left and right hand sides of assignment; the treatment is identical for values, names, and targets: lazy evaluation of the left hand side of assignment introduces zero complications.

In particular, if A and B return matrices; A returning a target, and B returning a value:

\[
\text{(assign} \quad (\text{transp} \ 0 \ \ (A) \quad (B) \text{)}
\]

and

\[
\text{(assign} \quad (A) \quad (\text{transp} \ 0 \ \ (B) \text{)}
\]

are equivalent, if A and B return square matrices.

2.1.12 Creating and accessing data-objects

There are three kinds of data-objects passed around in F-programs.

1. Data is passed anonymously up an F-tree as operands to arithmetic, geometric, or control F-functions. It is these that are subject to rank coercion; orientation; and shape and type coercion.

2. Data objects may be created using create, which also defines their scope, for side-effecting assignment. create also creates an association (an entry) in the dictionary, binding a identifier to the data-object.

3. Data may also be allocated and deallocated on heaps, using the F-functions global, dispose, mark, local. These are implemented in the F-code compiler, but are mainly
outside the subset of F-code considered in this thesis — and therefore do not receive a full treatment here. (Some details of heaps are given in the implementation of T-code, on page 166.)

The first two of these are of primary consideration for this thesis. Only create, hold (page 215), and template (page 215) affects the dictionary. In order to implement side-effecting assignments, the single F-function create is used to create and destroy objects. The definition of create is given on page 216.

For example:

```f-code
(create _a
  (B)
  integer
  (const 10)
)
```

Creates an integer array, whose identifier is _a, and whose length is 10. It then calls the F-tree B, returning the the result as the result of the create function. The result may be any data-parallel object. Then it destroys the array _a. The object will never be re-shaped once it is created. The scope of _a is the F-tree, B.

The create function is an eager function. _a is created, B is called, _a is destroyed:—synchronization is required to enforce the scope of _a.

B is potentially a lazy F-function, which includes arithmetic and geometric operations — and therefore does not impose any ordering restrictions between elemental operations. The conceptual chopping of F-programs into lazy and eager sections is shown in section 2.1.13.

Inside B, the variable _a may be read from and written to. Accessing the variable is by use of the F-function var, defined on page 217.

The following:

```f-code
(assign
  (var name _a)
  (var value _a)
)
```

is a pointless F-subprogram which does the assignment a:=a, by taking the name of _a as the left-hand-side, and the value of _a as the right-hand-side. The name and the value are both 10-element vectors because _a was defined to be a 10-element vector.

### 2.1.13 Lazy and eager sections

An F-code program is viewed as a collection of lazy-environments arranged in a tree which are divided by eager functions. This scheme is shown in figure 2.11. It shows eager functions as Ω, and lazy environments as Ξ. A lazy environment is a data-parallel section which possesses referential transparency, and can thus be executed in any order. Eager functions impose bulk-synchronization, but do not affect their primary operand and return it as a result. An example
of this type of function in F-code is `create`, which creates a data-parallel object, processes its operand (which may be a lazy section), destroys an object, then returns the value of the operand, without modifying it, as the result.

To demonstrate eager and lazy sections, the following program is used as an example:

```plaintext
(dyadic add
  (create A
    (comma right
      (assign
        (var name A )
        0 (const 1)
      )
      (dyadic add
        (var value A )
        0 (const 10)
      )
    )
  )
  integer
  (const 5)
)
(ramp
  (const 1)
  (const 5)
  (const 1)
)
)
```

This program first creates an integer vector of 5 elements, and associates it with the variable A. The left hand side of comma then assigns each element of the vector with the value 1. (The orientation mask for the assign means apply the scalar value 1, element-wise to vector A.) The program goes on to evaluate the right hand side of comma which adds the scalar 10 element-wise to A. Since the keyword of comma is right, the result of this addition is the result of the comma,
and hence the result of the `create`; this result is held in an implicit (anonymous) variable. The program then destroys variable `A`; and the result of the `create` is a five-element vector with each element being the value 11. Finally, the top addition adds the vector to a `ramp` (see page 220), which has the ramp of values 1 to 5 in steps of 1. The addition is carried out in another lazy environment. The result of this program is thus the five element vector containing the values: \(<12, 13, 14, 15, 16>\).

In this program, `create` is a \(\Omega\) the assignment is a \(\Xi\) environment: it does not matter in which order element-wise assignments are made. Both additions are \(\Xi\) environments: it does not matter in which order element-wise additions are made. The `comma` is part of the bottom add environment due to `comma`'s keyword. This whole scheme is shown in figure 2.12.

In order to demonstrate the ordering of this tree, the lines have been annotated with a letter:

```plaintext
a: (dyadic add
b:   (create A
    (comma right
c:     (assign
d:       (var name A )
e:         0 (const 1)
      )
f:     (dyadic add
g:       (var value A )
h:         0 (const 10)
```
Secondly, the events of creating and destroying variable A are named \( N \) and \( O \) respectively, which delimit the scope of variable A. In the expression below, square brackets are used to identify lazy environments, ‘;’ identifies sequentialization within a lazy environment, ‘!’ identifies possible parallel execution within a lazy environment, and ‘!’ identifies global (control) sequentialization. The ordering is then, implicitly: 
\[
N-iV\![(d|e);c]![(g|h);f]!O![(k|m);j; a]
\]

Lazy environments, such as \([(d|e);c]\) and \([(g|h);f]\) have no restriction of element-wise ordering. \( c,d,e,f,g,h \) identify data-parallel operations, or elemental operations which can be interlaced with respect to one another, and the ordering still holds elementally. The environments are equally \([(d_{e}|e_{z});c_{z}] \) and \([(g_{y}|h_{y});f_{y}] \), where \( e \) and \( y \) span across the index spaces of these F-subtrees and can be elementally executed independently.

A semantical restriction of F-code is that an eager function cannot be called elementally (cannot be called once for every elemental result). An eager F-function is called once, returning a data-parallel object which may be referenced elementally. The eager F-function in this program is \( b \). Its ordering is given, as above, as 
\[
N-iV\![(d|e);c]\![(g|h);f]\!O\![(k|m);j; a]
\]
Call all this \( B \). The top lazy environment requires this eager function, \( B \), to be evaluated before the lazy environment can be evaluated. The ordering of the top lazy environment is thus \( B\![(k|m);j; a] \). Which can be elementally executed \([(k_{s}|k_{z};m_{z});j_{s};a_{s}] \). An algorithm to provide the sequential ordering of an F-code tree — by visiting nodes in the F-tree in a particular, sequentialized order — is given later in the thesis, in chapter 4 (section 4.5.)

### 2.1.14 Functions in F-code

Functions/procedures, in the high-level-language sense, such as \( \text{foo}(X) \), can also be represented in F-code. The syntax for doing so requires the F-functions \( \text{const} \) (page 214), and \( \text{call} \) (page 231). Functions may be defined using the \( \text{const} \) function in the following way:

\[
(\text{const} (A))
\]

defines a function, whose body is the F-subprogram \( A \). \( \text{const} \) returns a pointer to this function, which can be written to a variable, such as \( _\text{foo} \):
The function itself is nameless (just a pointer), and may then be called using the variable:

\[(\text{call (var value } _\text{foo}) \text{ integer value 1})\]

Calls via the variable \( _\text{foo} \), returning a result which is integer, has rank 1, and is a value. The attributes are necessary for type inference. Functions may thus be defined anywhere inside an expression. call, hence F-code, does not permit arrays of functions: only one function may be called.

### 2.1.15 Arguments to functions, and the dictionary

The dictionary maintains associations to variables which are currently in scope. Functions which add new associations for variables to the dictionary are create, hold, and template. These are all eager functions: they add an association to the dictionary (an association with a piece of data, or a template); they proceed to evaluate their main operand (which may be a lazy environment); they finally destroy the association, and any data that was allocated.

Introducing the hold function of F-code:

\[(\text{hold } A \text{ (B) (C)}\]

Creates a variable \(A\), with the value \(C\), then executes \(B\), returning the result of the execution of \(B\) as the result of the hold (see page 215).

The high-level-program \(foo(X) + foo(Y)\) may be represented by the following F-code program:

\[(\text{comma right} \text{ (assign (var name } _\text{foo) (const (... foo body <parameter } $!>...) ) (dyadic add (hold } $1 \text{ (call (var value } _\text{foo}) \text{ integer value 2) (... X ...)) (transp 0 (hold } $1 \text{ (call (var value } _\text{foo}) \text{ integer value 2})\]}\]
2.1. INTRODUCTION TO F-CODE

foo is defined to be a function which takes parameter $\mathbf{X}$ and $\mathbf{Y}$ at each instance of the call. It exhibits the functional concurrency present in high-level programs. Recursion can easily be expressed in F-code:

\[
\begin{align*}
&\text{(comma right)} \\
&\text{(assign (var name \_foo))} \\
&\text{(const (call (var value \_foo) integer value 2))} \\
&\text{(call (var value \_foo) integer value 2)}
\end{align*}
\]

is the F-code representation for \texttt{foo} = \texttt{foo}, which is an infinite recursion.

2.1.16 Restrictions of the current implementation

1. Recursion is not implemented. This is an omission, and an area of future work.

2. Functions cannot be polymorphic. The compiler is still a PSP for most imperative high-level languages, in this respect.

3. The dictionary is implemented differently than as described in section 2.1.15. The thesis focuses mostly on the implementation of data-parallelism, without reference to interprocedural analysis. The need for a dynamic dictionary is eliminated. The scoping system adopted by this implementation is thus static, like C, FORTRAN and PASCAL which after all F-code was designed to support. The scoping system implemented is given in section 2.1.17.

4. A logical restriction of the semantics of F-code is that the discarded operand of \texttt{comma} should be an \texttt{assignment} only. This affects the compiler in one way: the compiler assumes that the discarded operand is an integer scalar value, which is a trivial type-check.

2.1.17 An implementation of F-code without a dictionary

The major interest in this thesis has been the implementation of lazy evaluation in a data-parallel setting, not the implementation of function calls and interprocedural analysis.

The current implementation of F-code is for a sequential machine, and there is no need to permit the parallel invocation of a function: \texttt{foo(X)} + \texttt{foo(Y)}^2 are sequentialized by this compiler. This is not a restriction, but the aim of the compiler. The one restriction is that arguments \texttt{X} and \texttt{Y} must be the same shape.
The dictionary is omitted by modifying the scoping system: parameters are always bound variables. In the original case, arguments were entries in the dictionary which were bound at a particular instance in time, around a call. It is the association at the time of the call which is an operand to a function.

\[ \text{foo}(X) + \text{foo}(Y) \]

can be implemented in the current compiler as: (which means that the semantics of F-code have not been implemented for function calls)

\[
\begin{align*}
\text{(create } & \$1 \text{)} \\
\text{(comma right)} & \\
\text{(assign)} & \\
\text{(var name } & \_\text{foo)} & \\
\text{(const } \ldots \text{ foo body } <\text{parameter } \$1> \ldots) & \\
\text{)} & \\
\text{(dyadic add)} & \\
\text{(comma right)} & \\
\text{(assign)} & \\
\text{(var name } & \$1) & \\
\text{(... X ...)} & \\
\text{)} & \\
\text{(call (var value } \_\text{foo) integer value 2)} & \\
\text{)} & \\
\text{(comma right)} & \\
\text{(assign)} & \\
\text{(var name } & \$1) & \\
\text{(... Y ...)} & \\
\text{)} & \\
\text{(call (var value } \_\text{foo) integer value 2)} & \\
\text{)} & \\
\text{)} & \\
\text{integer} & \\
\text{(const 10)} & \\
\text{(const 10)} & \\
\end{align*}
\]

This is the equivalent of passing parameters in global variables. This is adequate for the current implementation to a sequential target — but not adequate for a future implementation to a distributed target, which must, for function parameters (and variables whose scope is inside a function — automatic variables), use a dictionary. The dictionary for these variables cannot be implemented just on a stack:

\[
\begin{align*}
\text{(par)} & \\
\text{(hold } & \$1 \text{ (call (var value } \_\text{foo) \ldots} (A)) & \\
\text{(hold } & \$1 \text{ (call (var value } \_\text{foo) \ldots} (B)) & \\
\text{(hold } & \$1 \text{ (call (var value } \_\text{foo) \ldots} (C)) & \\
\end{align*}
\]

This invokes three simultaneous instances of a particular function. The parameter \$1 can have three simultaneous active instances (whose values are A, B, and C.) The dictionary cannot therefore be a stack, whereby the top instance is the active one. Also inside \_\text{foo}, there may be active instances of other variables. How this may be implemented is an area of future work:
the dynamic scoping mechanism of F-code makes it non-trivial. The solution seems to be to, on occasion, replicate the dictionary.

### 2.2 A quick guide to compiling F-code

This chapter has introduced informally the major concepts of F-code. A full definition of F-code is given in appendix A. The major, entirely bogus, criticism of F-code has been to date: "it cannot be compiled to run efficiently". And the major reason of this criticism is that it looks complicated, and looks unfamiliar. It is instead more general purpose and uniform than the new FORTRAN standards which purport to provide architecture-neutral platforms for parallel machines.
This section outlines the skeleton of the remainder of the thesis: implementing F-code. Figure 2.13 shows the architecture-independent stages of the F-code compiler (stages applicable to implementation on any architecture).

2.2.1 The front end of an F-code compiler, and inference (Chapter 3)

Chapter 3 describes the front end of the compiler. The front end consists of four primary phases: parsing, type-inference, shape-inference, and architecture-dependent optimization.

Introducing each stage of the front-end in turn:

Lexical Analyzer: This stage takes a ASCII based representation of F-code and turns it into a stream of tokens. Of course this is a very standard part of a compiler, see [ASU86]. At the moment, F-code is represented in ASCII text, but in order to cut down on the size of the file, it could be stored in an already tokenized (binary) file, and this stage would not be needed at all.

Parser: Using the stream of tokens, it is the job of the parser to construct a parse tree. It is also the job of the parser to create the symbol table, which is a table of identifiers (variable names) used in the program. The parser must be able to differentiate between different scopes. At this stage, the parse tree is very similar indeed to the original text, except it removes unnecessary symbols (such as lexical tokens like '(' and ')') and is much easier and faster to deal with.

In order to make the parsing algorithm more modifiable, it is done with the use of a parse table, such that the 'guts' of the process are easily seen, understood and changed.

At the moment, the parser does not deal with common-subexpressions, except for the smallest case which is all instances of the same identifier become a common-subexpression. It would be a useful addition for F-code to include a common-subexpression stage, however this may not strictly be architecture-independent.

Type Inference: In order to make the F-code representation more succinct, explicit type declarations are omitted from the source text. Using the newly formed parse tree, the type-inference stage infers the types of all nodes in the parse tree, and since it is the parse tree of a data-parallel language; it also infers the rank of each node; and due to the generic nature of geometric operations in F-code, also the sort.

The output of this stage of the compiler is then the parse tree, with attributes for type, rank and sort. The type-inference also adds to the parse tree explicit type-coercion nodes to identify the positions of type coercion for code generation. This means that the parse tree now is also strongly typed and monomorphic.

Type Checking: The type inference stage does not report any errors to the user, but rather tags positions in the tree where errors have taken place. The type-checker is quite a simple
pass of the tree (there have been two passes already: one to construct the tree, the second to infer the types). It is the job of the type-checker to inform the user of type-errors found in the type-inference stage to the user, accurately. One could argue that this stage of the compiler is not strictly necessary since F-code is an intermediate language and should therefore include no errors at all. One should assume that the compiler which produces F-code is error-free; the stage however is almost costless, and so it remains nonetheless. The output of the type-checker is no different than the output of the type-inference stage, if there are no errors. If there are errors, this stage abruptly terminates the compilation procedure.

**Shape inference:** This is the most complex part of chapter 3. One could argue that a sequential scalar language cannot be compiled unless it is strongly typed; similarly for data-parallel languages. On top of that, one could say that a data-parallel language cannot be compiled unless it is strongly shaped: that is that the shapes (the extents) of all operations are known at compile time. Similarly to inferring the types of all operations in the parse tree, it is necessary to infer the shapes of all operations in the parse tree. F-code includes a very rigorous definition scheme for specifying the shapes of all operations. The shapes of operations must be added as attributes to every node in the tree. They must be in place for later use in the compiler (especially for use in the code-generator, which must generate loops in the sequential case, or arrays of communicating programs in the parallel case, to execute to the data-parallel algorithm defined by the F-code text).

The compilation of F-code is not limited to programs in which the extents of operations are statically known to be constants. When they are not, the individual extents in a shape will be expressions involving run-time variables. The shape-inference stage in these cases constructs the expressions. In order to make the process orthogonal, these expressions are themselves represented by F-code parse trees, however in this case only scalar ones. This is given a thorough treatment at the end of the chapter; and it is the application of a novel technique. Inferring the shapes of expressions means that there is no need for an explicit dictionary to maintain shapes of objects. This is only true for the current implementation of the dictionary, however.

**Optimization:** One final part of the chapter deals with optimization — the aim of which is to improve the congruence of the implementation on a particular hardware platform. This is a very interesting future area of research, most applicable to distributed machines. The rudiments of machine-independent optimization are given in this thesis, but not yet implemented due to time-constraints, and the fact that the implementation in this thesis is not to a distributed machine, but to a single i860 system.
CHAPTER 2. INTRODUCTION TO F-CODE

2.2.2 Sequential aspects of F-code (Chapter 4)

An F-code program is composed of a number of eager and sequential sections. This chapter describes the sequential execution of F-code programs mainly on sequential machines, but also for asynchronous, distributed machines. Because this thesis is mainly aimed towards a scalar, sequential target, the diction of this chapter is aimed towards scalar machines: it talks of the implementation of loops rather than the distribution of data-parallel operations across arrays of processors, and partitioning of data. Many of the notions of this chapter are dualled on distributed machines. A discussion of this is given at the end of chapter 4.

It is too early in the thesis to summarize the sequentializing of F-code, but this chapter deals with the coarse-scheduling of F-code: the ordering which is necessary for any implementation of F-code between the eager and lazy sections of a particular program. Low-level (fine-grain) scheduling is not discussed until the chapter on targeting (chapter 6).

This chapter identifies an algorithm to traverse an F-tree recursively to produce a sequential visitation of eager sections and lazy sections. It visits the F-tree at a coarse level, only visiting the tops of these sections. Extra, internal, recursions at a fine level can be used to produce fine-grain scheduling. Nodes in the tree are labelled to identify lazy and eager sections using a process called loop filtering which is discussed in this chapter.

The chapter goes on to discuss the indices which are necessary to implement data-parallel geometric operations, orientation, etc. Part of the implementation strategy is to enumerate indices (infer what index does what) in an F-code program. This is shown to be a mechanical operation.

The aim of chapter 3 and chapter 4 is to prepare an F-tree for intermediate code generation and targeting which are the subjects of chapters 5 and 6 respectively.

2.2.3 Code generation (Chapter 5)

The compiler can be used to output a scalar C representation of an F-code program, which can be compiled by a compiler like gcc to produce a portable implementation of F-code without delving into low-level code generation at all. However, the intermediate representation should be one which does not discard the concurrency of the original F-code program in any form.

This chapter describes a low-level intermediate representation called T-code. T-code is a graph representation which can be used to target F-code to single RISC processors. T-code is akin to F-code in that it includes all of the functionality of F-code, without discarding concurrency — but at a lower level. The majority of machine-level optimization occurs at the T-code level. Nodes in the graph correspond to abstract RISC instructions which may or may not correspond to real RISC instructions: the abstract architecture is very orthogonal and has instructions which implement all of the intrinsical functions of F-code: cos, sine, etc. This orthogonality means that optimization can be included at the T-code level which does not need to take into account machine-specific details; the optimizations themselves can be orthogonal. An
example targetter (chapter 6) transforms T-code graphs into i860 instructions using a number of types of graph rewrite. After the targetter, further optimizations can be applied at the i860 level, which are very machine-specific.

While F-code is a data-parallel representation, T-code is purely scalar with loops to scan across data-parallel computations, like do loops. It is these loops which perform the selective evaluation of F-code. Iterations of the loop are independent; and the loop can be viewed as a sequentialization, or parallel execution across an index space, in just the same way as an F-code program. T-code graphs, however, have arcs which hold only scalar data (and pointers to data-parallel data). Coarse scheduling is achieved at this level with explicit par, and seq nodes. Coarse-level scheduling is explicit, unlike in the original F-code program.

A T-code graph is a graph which integrates control and data dependencies together. It can be traversed in any number of ways either following data-dependency arcs for register allocation etc. or following control dependency arcs for scheduling.

This chapter describes the process of transforming F-trees into T-code graphs. T-code graphs are produced, and at this point of the compiler, the F-code tree is discarded. T-code is unsuitable for high-level architecture-neutral and architecture-dependent data-parallel optimization. Conversely, F-code is inappropriate for low-level optimization and scheduling.

A T-code graph is the result of the front-end of the compiler. The targetter for a particular processor is a separate module, indeed program.

2.2.4 Targetting (Chapter 6)

This is the first stage of the compiler which is truly architecture-dependent. This is the aim of F-code and T-code: to delay architecture-specificness until the very latest stages of the compiler. This chapter discusses the process of transforming T-code graphs into i860 [graphs]. This is done using a small number of types of graph rewrite which expand T-code nodes, like macros, into i860 instruction nodes. The graph maintains the same form: control and data dependencies are integrated. Scheduling then proceeds by forming a spine (a linked list) of instructions from the graph, by choosing the most congruent execution schedule. The chapter discusses an iterative targetter which utilizes the arithmetic, memory, and execution pipelines of the i860, together with superscalar scheduling.

Figure 2.14 shows future and current work. At the moment the scheduler which has been coded is quite straightforward: it does little in the way of pipelining, and register allocation is also quite simple. There is a lot of literature about VLIW scheduling; but since such scheduling is beyond the time-scale of this thesis, VLIW scheduling is not addressed in this thesis, beyond a short discussion of it. The point is, the compiler gets to a point where VLIW scheduling can be carried out.

The iterative scheduler is projected work, discussed in chapters 6 and 7, which will give a more congruent implementation. The code quality without the iterative scheduler is already
similar to commercial compilers for the i860. This is a qualitative observation: commercial compilers often do no pipelining because the i860 is particularly difficult to pipeline.

Figure 2.14: Architecture-dependent compiler stages

The thesis shows the complete implementation of a compiler for an architecture-neutral platform from lexical analysis to congruent code generation. Chapter 7 gives some general conclusions about PSPs, F-code, and the approach taken to implement F-code.
Chapter 3

The front end of an F-code compiler, and inference

![Diagram of front end compiler phases]

Figure 3.1: Front end compiler phases
3.1 Introduction

Of course, writing any compiler is a long and convoluted process; and its problems and techniques must be introduced gradually if they are to be easily or properly understood by a reader of this thesis. Low-level implementation details are generally soporific to read, and so low-level details are omitted or relegated to appendices. Likewise, where possible, algorithms are demonstrated rather than described. Fortunately compilers fall roughly into stages, and compilation techniques are independent. As a guide to the reader, this means if a technique cannot be thoroughly understood, merely take as read that it does what it is supposed to do, and move onto the next. It is unlikely that a reader will understand all aspects of the compilation process, unless s/he is implementing a similar compiler.

As with any compiler, the F-code compiler is a mapping between an input source language and some machine or intermediate language. The diagram on the previous page gives a guideline to the stages involved in compiling from F-code source to an intermediate stage of compilation, in which all the stages are still absolutely machine independent.

Implementationally, it is very useful if these stages of the compilation process are written in ways that are easily modifiable: the structure of the compiler will never be radically changed, but F-code is a prototype language only, and it will in time be modified and perhaps extended. One can also add that F-code as it stands is a very suitable medium for representing data-parallel algorithms, and its extensions and modifications should mostly be minor, since this is the nature of the problem.
3.2 Lexical analysis and parsing

The syntax of F-code is quite 'Lisp-like'; lexical analysis and parsing are particularly easy. Lexical analysis is the process of turning an input stream (normally of characters) into tokens. Parsing is the process of syntax analysis, and creating a tree. The current representation of F-code is text based; the file representing it can be pre-tokenized and represented by a binary file.

3.2.1 Lexical analysis

The lexical analyzer maintains a set of token classes, corresponding to the BNF grammar of F-code given in the previous chapter.

```
CLASSES = { LEX, LEXCONST, TYPE, SORT, ACCESS, PROPERTY, SELECTOR, FUNCTION, UNARY, BINARY, TOTAL, INT, NUMBER, MASK, IDENTIFIER }
```

F-code has no redundant syntactic sugar since it is an intermediate language, and thus the table of lexical tokens is particularly small and the parser is relatively straightforward.

```
<table>
<thead>
<tr>
<th>Class</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEX</td>
<td>(, )</td>
</tr>
<tr>
<td>LEXCONST</td>
<td>'Top', 'Epsilon', 'Max-Int', 'Max-Char', 'Hole'</td>
</tr>
<tr>
<td>TYPE</td>
<td>'Logical', 'Character', 'Integer', 'Real', 'Complex'</td>
</tr>
<tr>
<td>SORT</td>
<td>'Name', 'Value', 'Target'</td>
</tr>
<tr>
<td>ACCESS</td>
<td>'Name', 'Value', 'Target', 'Pointer'</td>
</tr>
<tr>
<td>PROPERTY</td>
<td>'Name', 'Value', 'Target', 'Couple'</td>
</tr>
<tr>
<td>SELECTOR</td>
<td>'Re', 'Im'</td>
</tr>
<tr>
<td>DIRECTION</td>
<td>'Left', 'Right'</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>'Const', 'Hold', 'Template', 'Create', 'Var'</td>
</tr>
<tr>
<td></td>
<td>'Global', 'Dispose', 'Local', 'Mark', 'Ramp'</td>
</tr>
<tr>
<td></td>
<td>'Dyadic', 'Choice', 'Reduce', 'Transp', 'Sect'</td>
</tr>
<tr>
<td></td>
<td>'Repl', 'Pack', 'Gather', 'Diag', 'Transform'</td>
</tr>
<tr>
<td></td>
<td>'Comp', 'Pol', 'Displace', 'Distance', 'Type'</td>
</tr>
<tr>
<td></td>
<td>'Assign', 'Channel', 'Pat', 'Get', 'Seq'</td>
</tr>
<tr>
<td></td>
<td>'Par', 'Loop', 'Spawn', 'If', 'Call'</td>
</tr>
<tr>
<td></td>
<td>'Select', 'Monadic', 'Slice', 'Part', 'Shape'</td>
</tr>
<tr>
<td></td>
<td>'Comma', 'Coerce'</td>
</tr>
<tr>
<td>UNARY</td>
<td>'Neg', 'Inv', 'Frac', 'Sqrt', 'Arg'</td>
</tr>
<tr>
<td></td>
<td>'Modulus', 'Sin', 'Cos', 'Tan', 'Arcsin'</td>
</tr>
<tr>
<td></td>
<td>'Arctan', 'Sinh', 'Cosh', 'Tanh', 'Exp'</td>
</tr>
<tr>
<td></td>
<td>'Even', 'Odd', 'Not', 'Bit-Not', 'Round'</td>
</tr>
<tr>
<td></td>
<td>'Font', 'Conj', 'Arccos', 'Ln', 'Trunc'</td>
</tr>
<tr>
<td>TOTAL</td>
<td>'Add', 'Mul', 'Max', 'Min', 'And'</td>
</tr>
<tr>
<td></td>
<td>'Or', 'Xor', 'Bit-And', 'Bit-Or', 'Bit-Xor'</td>
</tr>
<tr>
<td></td>
<td>'Sub', 'Div', 'Idiv', 'Modulo', 'Pow'</td>
</tr>
<tr>
<td></td>
<td>'lpow', 'Log', 'Shr', 'Shl', 'Ror'</td>
</tr>
<tr>
<td></td>
<td>'Rol', 'Gt', 'Ge', 'Lt', 'Le'</td>
</tr>
<tr>
<td></td>
<td>'Eq', 'Ne'</td>
</tr>
<tr>
<td>INT</td>
<td>Integers eg.— 10 30 0</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Numbers eg.— 12 5.7 10+5i</td>
</tr>
<tr>
<td>MASK</td>
<td>Masks eg.— 010 1100 00</td>
</tr>
<tr>
<td>IDENTIFIER</td>
<td>Identifiers eg.— Yabba. Dabba $d.o.o$</td>
</tr>
</tbody>
</table>
The lexical analyzer has one parameter which is a set of permissible classes at any point in the parse. E.g.: lex(TYPE | IDENTIFIER) consumes either a type name or an identifier. This is equivalent to MODE in the grammar for F-code. The IDENTIFIER class is always the last checked. Lexical and syntax errors are detected if the token being read is none of the permissible classes.

See appendix B for the lexical table. The parser returns an index into this table which is a number. This gives three handles on the token: the string (for pretty-printing), the token value and the token class. E.g.: lex(UNARY) produces some value i. T_N(i) is the string, which may be "Neg", if "neg" was parsed in the input stream. T_V(i) is the value NEG, and T_C(i) is the value UNARY.

3.2.2 Parsing

YACC is not used to implement the front-end of the compiler because it is unnecessary and would only hinder easy modification to the compiler: the parse-table is also used to contain type inference rules.

Although lexical analysis and parsing are very well known parts of a compiler, it is useful to note their implementation for accessibility. It would be attractive to make the, although simple, grammar for F-code modifiable. It is also impossible to describe type inference without first describing parsing, since the two are related.

The best way to do this is to use a table-driven approach. The full grammar is written in terms of a table (which represents a graph). Then a start state is chosen, which in the graph of appendix C is t_expr. To represent a BNF grammar for LR parsing in table form, three types of table entry are needed:

1. The ability to consume a token (and build the parse tree).
   A parse tree which has only three (or less) arcs.
   The grammar $NAME ::= $TOKEN $LEFT $RIGHT
   can be represented by a table entry
   $NAME ( $CLASS, $VALUE, $LEFT_NAME, $RIGHT_NAME )
   It consumes a token $TOKEN, and then goes on to parse using $LEFT_NAME and then using $RIGHT_NAME.
   A generic F-code expression is defined by the two following entries in a table:
   t_expr ( LEX, OPEN, t_funcs, t_close )
   t_close ( LEX, CLOSE, _, _ )
   That is, an expression is started with a token of class LEX and value OPEN, which is the symbol '('.
   It is followed by parsing using the table t_funcs, and is finished using a token of class LEX and value CLOSE, which is the symbol ')'.

2. Choice
   If an entry $NAME ( $CLASS, $VALUE, $LEFT_NAME, $RIGHT_NAME ) is consid-
3.2. **LEXICAL ANALYSIS AND PARSING**

It only matches the current lexical token, if the class of the token is the same as the $CLASS$ of the entry and the value of the token is the same as the $VALUE$ of the token. The table entry for $VALUE$ can be a wild card, denoted by the symbol "*". This matches any value of the lexical token. Choice is represented by having more than one entry with the same $NAME$. In order to make this simpler, the parser is given an ability to look-ahead for tokens.

By example, the implementation of $t\_exprs$ (any number of expressions) is:

```
t\_exprs ( TEMP | LEX, CLOSE, _, _ )
t\_exprs ( TEMP | IDENTIFIER, _, _, _ )
t\_exprs ( TEMP | TYPE, _, _, _ )
t\_exprs ( TEMP | LEX, OPEN, t\_expr, t\_exprs )
```

These are checked (in order) by the parse automaton. The first three are merely checks to see when there are no more $t\_exprs$.

The last is the most important: $SEXPRS := \text{EXPR} \ \text{SEXPRS}$. The use of $TEMP$ in these rules means look-ahead. Do not consume a lexical token.

3. The last type of rule is one which matches regardless of the CLASS or VALUE of a token:

```
t\_expr2 ( _, t\_expr, t\_expr )
```

It does not consume a token, but creates a binary node, on the left is $t\_expr$ and on the right is $t\_expr$.

The table also needs to take into account scoping. There are definite scopes for variables in F-code, and simple additions to the table create variables (and hide other variables with the same name). The symbol for this is $HIDDEN$. The full grammar for F-code is given in appendix C.

One can imagine the parse tree as being built as the parse automaton traverses the parse table. Each entry it successfully visits creates a node in the parse tree. The node in the parse tree maintains a pointer to the entry which created it, and the type of the node is the current value of the index into the lexical table. If the entry in the parse tree is one which does not need to call $lex$, it is given a special type — $BIN$, meaning binary node.

There are a number of optimizations which are done to this. If the entry is of a type which looks ahead and does not consume a token, a parse tree node is not created. There is also no necessity for nodes in the tree which are of LEX type. '(' and ')') are not needed since they are represented implicitly by the shape of the tree. Similarly the F-code symbols $CONST$, $DYADIC$, $MONADIC$, and $REDUCE$ do not need nodes since they can be implied by their operation. For example, $(REDUCE \text{MAX} \ .\ )$ can be represented by a single node which is $\text{MAX}$, and whose table entry suggests that it is a reduce operation. In most cases the shape of the trace through the parse table is the shape of the parse tree.
There are no provisions for common subexpressions yet, except for identifiers. All instances of an identifier in a given scope are a common subexpression. Any inference assertion to any instance of an identifier is then naturally asserted to all instances of that variable.

An example of table-based parsing is given below, and in figure 3.2. The figure has four parts: the original F-code source text, the arcs used in table based parsing, the final parse tree, and the traversal through the parse tables. Having already found an open bracket, the parsing starts at the top of the trace through the parse tables. The next token is dyadic, allowing the parser to move via arc 1, the next token is max, allowing the parser to move via arc 2, and so on ... Only some of the items visited in the parse table make nodes in the parse tree, and so we see that the parse tree is somewhat simpler than the text.
3.3 Type inference

F-code does not include any but necessary explicit type information. The basic reason for this is that it keeps the representation succinct. Type inference is a quick and relatively costless procedure. In order to compile F-code, a program needs to be made monomorphic by inferring the types of every node of the tree.

Each node in the parse tree retains a pointer to the parse table entry which created it. The parse table entry also includes a set of type inference rules to be applied to this kind of node in a parse tree. All type inferences are local. Most type inferences occur between adjacent functions in a tree. Those inferences which appear to be non-local — for example those between create and var which create and reference a variable — are rendered local by the parser, which explicitly creates links between the two.

3.3.1 Type information

Type inference adds to every node of the parse tree type information in the form of an inference $I_i$. A type inference $I_i$ is defined as a triple: $I_i: (\text{type}, \text{rank}, \text{sort})$, where

- **type** is a subrange of the ordered set:
  
  \{\text{logical, character, integer, real, complex}\} The full set represents the case when type is unknown. The types form an ascending hierarchy in the same order as they are listed. If at any stage type $=$ $\emptyset$ the enclosing F-expression cannot be consistently typed.

- **rank** $\in \mathbb{Z}^+ \cup \{\tau\}$, where $\tau$ is used to represent the case when rank is unknown, and $\mathbb{Z}^+$ is the set of non-negative integers.

- **sort** $\in \{\text{name, value, target}\} \cup \{\tau\}$, where $\tau$ is used to represent the case when sort is unknown.

The three attributes for a type inference are independent. The default type inference for a node, at the point when it is created, is $I_i : (\text{logical, character, integer, real, complex}, \tau, \tau)$

3.3.2 Type inference procedure

The task of type-inference is to tighten the default inference to some unambiguous instance of type, using all available type inference rules. The type-inference of a node is a triple, including the type, rank and sort of the node.

An expression is only fully inferred when type-inferences of all nodes in the tree are degenerate (unambiguous). A degenerate inference is one such that $\#(\text{type}) = 1$, rank $\in \mathbb{Z}^+$, sort $\in \{\text{name, value, target}\}$ or in other words when $I_i$ is unambiguous, and the tree is monomorphic. When the tree is monomorphic it can be compiled without run-time checks.

Four basic classes of type inference exist: Given $N_i$ is the current node for the inference procedure:
1. An attribute of $I_i$ can be determined from $N_i$ directly.

2. An attribute of $I_i$ can be determined from the type inference of a number of $N_i$'s subnodes.

3. The attributes of the type inference of $N_i$'s subnodes can be determined from $N_i$.

4. Non-local inferences can be made by using pointers created by the parser. Non-local inferences always occur via variables. The point of reference to a variable: var maintains a pointer to the node which creates the scope for this variable, such as hold, or create. The type of the var can therefore be inferred from the point of creation. Attributes of $I_i$ are determined from the point of creation of the variable referenced in $N_i$.

One or more of these classes of inference may be made at any particular node. Each inference tightens the inference of a node:

- $\text{rank}$ may be re-fixed ($\text{rank}$ is originally $r$), $\text{rank} \in Z^+$, iff $\text{rank}$ was previously $r$. An attempt to re-fix the rank from another value is an error, because it would mean that the rank of this function is inconsistent in the program. In such a case $\text{rank}$ is left as it was beforehand. A rank mismatch node will be added to the tree at precisely the point of the error. This is detected by a later type checking pass of the tree. The rank mismatch node has two attribute which are the line numbers of the original F-code program between which the error occurs. This presumes a certain textual layout style for F-code programs — one function per line (this layout is assumed for all demonstration programs in this thesis). This method of reporting errors is not particularly sophisticated, and is only a preliminary one.

- $\text{sort}$ may be re-fixed, $\text{sort} \in \{\text{name, target, value}\}$, iff $\text{sort}$ was previously $r$. An attempt to re-fix the sort from another value is an error, because it would mean that the sort of this function is inconsistent in the program. In this case $\text{sort}$ is left as it was beforehand. A sort mismatch node will be added to the tree at precisely the point of the error. This is detected by a later type checking pass of the tree.

- $\text{type}$ may be limited to some subset (normally a subrange) of its current value. If this becomes the emptyset, this is a type mismatch. A type mismatch node will be added to the tree at precisely the point of the error. Later in the inference procedure, if the mismatch is between unique types, the type mismatch node is transformed into a type coercion node. Type mismatch nodes are detected by a later type checking pass of the tree. Type coercion nodes (which are not part of the external definition of F-code) pass through the type checker without report.
3.3.3 Examples of type inference

This transformation will be demonstrated with a number of simple examples. In each of these, capital letters denote arbitrary F-code expressions with previously inferred attributes; attributes are written inside braces; neither of these are part of the syntax of F-code, but are written this way for didactic purposes. Type inference is the process of percolating types up and down the tree simultaneously and placing type coercions wherever conflicts occur between types.

```
( dyadic add
  ( {int} A )
  ( {int} B )
) --------->
( {int} dyadic add
  ( {int} A )
  ( {int} B )
)

( dyadic add
  ( {real} A )
  ( {real} B )
) --------->
( {real} dyadic add
  ( {real} A )
  ( {real} B )
)

( dyadic add
  ( {real} A )
  ( {int} B )
) --------->
( {real} dyadic add
  ( {real} A )
  ( {real} type_coerce
    ( {int} B )
  )
)
```

Those were examples of inference occurring up the tree — from A and B up to the add. There are functions passing type information down the tree, for example the operation \texttt{part} takes the real or imaginary part of a complex number, resulting in a real number:

```
( part re
  ( A )
) --------->
( {real} part re
  ( {complex} A )
)

( part re
  ( {real} A )
) --------->
( {real} part re
  ( {complex} type_coerce
    ( {real} A )
  )
)
```

Finally, non-local inferences may occur via \texttt{var} functions in F-code. Consider the following:

```
( create \_x
  ( ....
    ( var value \_x )
  )
  integer
  (const 10)
)
```

```
( create \_x
  ( ....
    ( {integer} var value \_x )
  )
  integer
  (const 10)
)
```

This inference is non-local, the attribute \texttt{integer} is propagated to the \texttt{var}. This occurs via a pointer in the symbol table, at the entry for \_x, to the \texttt{create} node. The \texttt{integer} keyword from the \texttt{create} becomes the type of the \texttt{var}. 
3.3.4 Examples of rank and sort inference

Rank (and sort) inference is more trivial — the number in the inference is the rank. The binary masks in the following examples denote active dimensions for an operation. 00 is applicable to a constant because it has no dimensions; 01 orientates a vector along the first axis of a matrix; 10 orientates a vector along the second axis of a matrix; and so on. Ones denote active dimensions.

\[
\text{for } (i=0; i<\ldots; i++) \text{ for } (j=0; j<\ldots; j++) \text{ result}[i,j] = A[i] + B[j];
\]

is written in F-code: (dyadic add 10 (A) 01 (B))

Taking the maximum of two vectors:

\[
(\text{dyadic max} \quad \rightarrow \quad \{\text{int},1,\text{val}\} \text{ dyadic max})
\]

\[
(\{\text{int},1,\text{val}\} A) \quad (\{\text{int},1,\text{val}\} B)
\]

Taking the maximum of a matrix and the constant value 4:

\[
(\text{dyadic max} \quad \rightarrow \quad \{\text{int},2,\text{val}\} \text{ dyadic max})
\]

\[
00 (\{\text{int},0,\text{val}\} \text{ const 4}) \quad 00 (\{\text{int},0,\text{val}\} \text{ const 4})
\]

\[
11 (\{\text{int},2,\text{val}\} B) \quad 11 (\{\text{int},2,\text{val}\} B)
\]

Adding two vectors, orientated to make a matrix:

\[
(\text{dyadic add} \quad \rightarrow \quad \{\text{real},2,\text{val}\} \text{ dyadic add})
\]

\[
01 (\{\text{real},1,\text{val}\} A) \quad 01 (\{\text{real},1,\text{val}\} A)
\]

\[
10 (\{\text{real},1,\text{val}\} B) \quad 10 (\{\text{real},1,\text{val}\} B)
\]

A full definition of type inference rules applied to F-code is given in appendices D and E. The major result of this section is that without any but necessary type information, the rest can be inferred in any consistent F-code program. The tree is rewritten during type inference to include all type-coercions, explicitly. This allows the rest of the compiler to work with a consistent, monomorphic tree. Type coercions are inserted to make code generation easier: coercions now have specific positions.

3.4 Type checking and type inference ambiguities

Since F-code is an intermediate language a type checking stage is not strictly necessary, however this, and compilers targeting to F-code, is and are prototype compilers. It is therefore invaluable for debugging.

The type checking stage has two functions, and should be seen as the closing part of the inference stage. It merely has to scan the tree and check for mismatch nodes (those conflicts which cannot be resolved); and produce error messages. Of course error messages cannot be created as soon as conflicts arise because, later, type conflicts may be resolved by making them
3.5 E V A L U A T I N G C O N S T A N T S U B-E X P R E S S I O N S

Type coercions. It is thus necessary to include in the node, the line number in the original F-code program. The type checker can thus produce 'meaningful' error messages.

The second motive for the type checking stage, is that it completes the job of type inference for some, relatively obscure, F-code programs which cannot be inferred to be unambiguous: The type checking pass clears away some type inference problems. A difficulty occurs when all nodes in the F-code tree cannot be made degenerate and their types remain ambiguous. These cases are a natural consequence of the omission of type information from an original program:

A problem occurs for functions like monadic neg, whose a priori types are a range. If one takes, for example, monadic neg of a character value, it creates a type-mismatch from char to an uncertain type which is between integer and complex. The range is not limited any more during the inference procedure, since it may introduce unnecessary type coercions further up the tree. The type-checker has the additive quality of choosing the integer alternative, or actually on the i860 it is appropriate to use the real alternative, since integer operations are more costly! Ambiguities are eradicated during the type checker by taking the minimum type of those possible for values, and the maximum type of those possible for targets. Besides these cases, an F-code program is always inferrable to be monomorphic unambiguously.

While independent of type-inference, and type-checking, during the type checking stage, constant folding is applied to fold constant subexpressions to single constant values. This, at the moment, is only applicable to scalar values; in future it may be appropriate to extend this to contend with vectors, or higher-ranked objects. This is not a limitation of the compiler, but of F-code as a language. F-code should be changed such that it is possible to declare non-scalar constants, since at present it only allows scalar constants, and strings (vector characters). The only purpose of this, of course, is to reduce the amount of unnecessary computation. Constant folding may already have been done in the front-end compiler; it is strictly unnecessary for a front-end compiler to do so.

3.5 Evaluating constant sub-expressions

During the parse stage of the compiler, constant (scalar) values are marked with an extra 'constant-bit'; this is also true of constant (scalar) values which are created at any time — for example, while the tree is being re-written.

<foldables> ::= monadic | dyadic | {type coerce} | comma | part

Only these need to be evaluated. The algorithm is obvious: it merely checks the 'constant-bit' of both subordinates for these functions, evaluates the result and re-rewrites (folds) the tree. There is an added complication which will be given in section 4.4.4 of chapter 4.

Only scalar constant sub-expressions are dealt with. It is more than feasible, if a new version of the F-code function const is introduced which can represent arrays, etc. of values that this can be extended to non-scalar sub-expressions.
3.6 Interpreted F-code

F-code can indeed be interpreted quite easily, without prior type inference because such an implementation could detect and apply itself to type inference information inferred at run-time. Nor, at little extra relative overhead, need a program be monomorphic.

The type inference stage is useful in cases even when F-code is to be interpreted. It is not, however, in any way necessary. The purpose of type inference is to make run-time checks for type, unnecessary. It would be possible to replicate sections of code for different types and use a polymorphic implementation. This ordinarily has far too high a memory, or efficiency, cost. It is also not out of the question to consider polymorphism in terms of operation rank, however this goes against the grain of the operand orientation ideas of F-code and is unheard of in imperative programming languages. F-code programs which are compilable are those which are purely monomorphic. The type checker detects all type-inference errors of this sort and compilation is aborted immediately after the type checker stage.

3.7 Shape inference

The last sections described how a fully type-inferred F-code tree was obtained. The next stage of the process of compiling F-code, is to infer the extents of every operation in the F-tree, such that it is compilable.

The array of extents of a node in F-tree is the shape: a 4 x 5 matrix has the shape [4, 5]. There is a need to create such arrays for every node of the tree. It has to be for every node since the extents of a node almost always depends on those of it's subordinates, and when it does not it is in terms of a non-local shape inference.

The shape, or extents, of a node may not be known at compile-time since objects in F-code can have dynamic shape. In this case, the extent of an operation may be an expression including run-time variables. Shape inference is the procedure to create these ancillary expressions, or constant values if extent is known at compile-time. The expressions are themselves, as is demonstrated later, scalar F-expressions. Where dynamic shape is instituted, it does not decimate run-time efficiency (though it will affect it), but does increase the complexity of the compilation and run-time task, especially when F-code is targetted to a distributed machine. This will, as any program will, if efficiency is any aim at all, require run-time re-distributions and re-alignments. At this stage of the compilation algorithm only architecture-neutral details are considered. Dynamic shape is a very important part of a language, and should not be omitted.

See sections 2.1.8 and 2.1.9 for details of the rank coercion and shape coercion mechanisms; full details in [BMS92]. Shape inference is done in a bottom-up manner, since the shape of a node in the F-tree usually only depends on the shapes of daughter nodes. It may, in the case of var depend on the parameters of the node which created a variable: for F-functions like create, or hold. These are equally subordinate nodes to the var, since the parser creates pointers to
the point of creation of a variable. Daughters are also 'subordinates'.

Before delving into details of the algorithm, the aim of shape inference is to create ancillary shape expressions, which are scalar F-code expressions which evaluate the extents of data-parallel F-code operations. Figure 3.3 shows examples of inferred shape expressions.

The left example creates a vector object $A$ of extent 10. Therefore every reference to it yields the extent 10. The length of the ramp in the left example is 5. Shape coercion therefore dictates that the length of the addition is 5 (since this is the minimum of 5 and 10). This extent inferred for the create and comma are identical to the extent of the add.

The right example is an example of inferring the extents of a purely geometric operation. The shape expressions it creates reflect orientation masks, and the transposition is a 'x' in the shape expressions.

After shape-inference, for every dimension of every node in the primary F-code tree, is a F-expression which gives the extent. While the items in the shape expressions here are shown as numbers, they are really the F-code instruction (const n). In the examples, all of the extents are statically known at compile-time and so they are all folded to be constants. The arrows in the shape-expressions represent dependencies between indices: this will be shown in the next chapter. This section goes on to describe the process of shape inference.
3.7.1 Notation for F-code with inferred shape

In the following examples, shapes, being arrays of extents, are denoted inside square brackets before the name of any function in the F-tree. For example, if a function returns an object with shape 4 x 5, its shape will be \([(\text{const 4}), (\text{const 5})]\).

Individual extents are integer scalar F-expressions, in all cases; since an extent is always integer and always a scalar. Where extents are not known at compile time, such extent expressions will include \text{var}. See page 217 for the definition of \text{var}. They will always be of the sort \((\text{var value } X)\), where \(X\) is an identifier. To make the notation shorter and clearer the keywords \text{const} and \text{var value} are omitted.

\([4,5]\) is the same as \([(\text{const 4}), (\text{const 5})]\) and represents the shape of a 4 x 5 object.

\([_x0, _x1]\) is the same as \([(\text{var value } _x0), (\text{var value } _x1)]\) and represents the shape of an \(_x0 \times _x1\) object.

This notation is not part of the F-code definition and cannot be parsed, but it is useful to show examples, as follow.

3.7.2 Shape inference examples

Example 1 — simple example

If the inference for \(Y\) is considered, in the following simple program (which adds two vectors together):

\[
\begin{align*}
(Y) & \text{ dyadic add} \\
 & \begin{align*}
 & ([X1] A) \\
 & ([X2] B)
\end{align*}
\end{align*}
\]

The shape rule for \text{dyadic add} is \(d = \min (e', e'')\). See page 221.

Then \(Y\) is the F-expression \((\text{dyadic min } X1 X2)\). It inherits \(X1\) and \(X2\) as common sub-expressions. This expression for \(Y\) may subsequently be folded if \(X1\) and \(X2\) are both constants.

These extents are ancillary to the original and main F-expression, and are associated with the original tree. They are always scalar values; and are usually integer expressions.

\(X1 = (\text{const 4})\)
\(X2 = (\text{const 5})\)
\(Y = (\text{dyadic min } X1 X2)\)
\(= (\text{dyadic min } (\text{const 4}) (\text{const 5}))\)
\(= (\text{const 4}) / \text{After folding}\)

\(X1 = (\text{var value } _x1)\)
\(X2 = (\text{const 5})\)
\(Y = (\text{dyadic min } (\text{var value } _x) (\text{const 5})) / \text{Cannot fold}\)

Shape inference is done on the return journey of a recursion through the original F-tree since shape expressions are always constructed from subordinates.
3.7. SHAPE INFERENCE

Example 2 — orientation

Secondly, a slightly more complicated example, including operand orientation which adds a vector to a matrix in a particular orientation. Since A is a vector, it has a single extent X1, and since B is a matrix it has two extents X2 and X3. It results in a matrix whose extents are Y and Z.

\[
([Y,Z] \text{dyadic add } 01 ([X1] A) \\
11 ([X2,X3] B))
\]

The same shape rule is used: \( d = \min(e^r, e^r) \). However the expansion \( e^r \) includes an \( \infty \). Then \( Y \) is the \( F \)-expression (dyadic \( \min \infty X2 \)) and \( Z \) is the \( F \)-expression (dyadic \( \min X1 X3 \)). There is no sense in creating the expression for \( Y \) in this form, and so it is instead written (monadic ref \( X2 \)), which is equivalent.

\[
\begin{align*}
X1 &= (\text{const} 4) \\
X2 &= (\text{const} 5) \\
X3 &= (\text{const} 6) \\
Y &= (\text{monadic ref} X2) \\
&= (\text{monadic ref} (\text{const} 5)) \\
&= (\text{const} 5) \text{ / After folding} \\
Z &= (\text{dyadic min} X1 X3) \\
&= (\text{dyadic min} (\text{const} 4) (\text{const} 6)) \\
&= (\text{const} 4) \text{ / After folding}
\end{align*}
\]

Example 3 — comma

The definition of comma is on page 229. In the examples, the value of \( X \) is inferred.

\[
\begin{align*}
([X] \text{comma left} \\
&([Y] L) \\
&([Z] R))
\end{align*}
\]

\[
\begin{align*}
([X] \text{comma right} \\
&([Y] L) \\
&([Z] R))
\end{align*}
\]

Comma always evaluates the left expression and then the right expression. This function can be used to sequentialize two activities which yield some data object to be processed further, and which for that reason can not be placed in a seq list. When sequentialization is not needed, function choice may be used instead. In the first of these, \( X = (\text{monadic ref} Y) \) and in the second, \( X = (\text{monadic ref} Z) \).
Example 4 — ramp

The definition of ramp is on page 220. In the example, the value of \( X \) is inferred.

\[
([X] \text{ ramp}
\begin{array}{c}
([[]] B) \\
([[]] E) \\
([[]] S)
\end{array}
)
\]

The shape rule is: \( d_0 = \left[ \frac{a^2 - b^2}{c^2} \right] + 1 \). Then \( X \) is the F-expression:

\[
\begin{array}{c}
\text{(dyadic add)} \\
\text{(monadic abs)} \\
\text{(dyadic idiv)} \\
\text{(dyadic sub} E \ B) \\
S
\end{array}
\]

This is the first example where the extent depends directly on the tree itself. If \( E, B \) or \( S \) are not constants, then this F-expression cannot be folded to a constant.

\[
\begin{align*}
B &= (\text{const 1}) \\
E &= (\text{const 10}) \\
S &= (\text{const 1}) \\
X &= (\text{const 10}) /\text{fully folded}
\end{align*}
\]

Section 3.7.3 deals with cases where an (ancillary) extent tree depends directly on the original (main) F-code tree.

Example 5 — transform

The definition of transform is on page 225. In the example, the value of \( Y \) and \( Z \) are inferred.

\[
([Y, Z] \text{ transform}
\begin{array}{c}
([\ldots, \ldots] A) \\
([X1, X2] T1) \\
([X3, X4] T2)
\end{array}
)
\]

The shape rule is: \( d = \text{min}_{0 \leq i < n} e^i \). Then \( Y \) is the F-expression \((\text{dyadic min} X1 X3)\) and \( Z \) is the F-expression \((\text{dyadic min} X2 X4)\). The extents of \( A \) do not matter at all, except the rank of \( A \) must be the same as the number of index operands.
3.7. SHAPE INFERENCE

Example 6 — create

The definition of create is on page 216.

\[
([Y_1,Y_2] \text{ create } A
\]
\[
([S_1,S_2] \text{ } S)
\]
\[
(\square X_1)
\]
\[
(\square X_2)
\]
\[
(\square X_3)
\]

The inference for create is trivial, since create does nothing to its operand \( S \). Then \( Y_1 = (\text{monadic ref } S_1) \) and \( Y_2 = (\text{monadic ref } S_2) \). The variable \( A \) is valid for the scope of expression \( S \).

Example 7 — var

The definition of var is on page 217. In the Example 6, for create, it does nothing to infer anything about the shape of variable \( A \). This is only done when the variable is referenced. When a create function is parsed, the dictionary entry for symbol \( A \) maintains a pointer to the F-function which creates its scope. In the above example, it was to the create. All occurrences of an identifier are, turned by the parser into common sub-expressions. The two \( A \)s in the following program are the same node in a parse tree:

\[
(\text{create } A
\]
\[
\text{...}
\]
\[
([Y,Z] \text{ var value } A)
\]
\[
\text{...}
\]
\[
\text{integer}
\]
\[
(\square X_1)
\]
\[
(\square X_2)
\]

The identifier \( A \) maintains a pointer to the F-function which created it, which is the create. Using this pointer, one would infer the expressions for \( Y \) and \( Z \). These are easy to infer (although the inference is non-local) to be \( Y = (\text{monadic ref } X_1) \), and \( Z = (\text{monadic ref } X_2) \). These are actually references to the original F-code tree, and similar to Example 4, for Ramp, they may require the original tree to be rewritten. See section 3.7.3 for the case where the shape \([Y,Z]\) is dynamic.

Example 8 — comp

The definition of comp is given on page 225.

\[
([Y,Z] \text{ comp } 0
\]
\[
([X_1,X_2] \text{ } A)
\]
\[
([X_3,X_4] \text{ } B)
\]

The shape of the result is defined as \( d_j = \begin{cases} \min(e_j^1, e_j^2), & \text{if } j \neq N \\ \phi^j + \phi^r, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(e') \). Then it is easy to infer that \( Y = (\text{dyadic add } (X_1) (X_3)) \) and \( Z = (\text{dyadic min } (X_2) (X_4)) \). This is because \( Y \) is active when on the \( N^{th} \) dimension, and \( Z \) is not.

\[
\begin{align*}
\{Y, Z\} \text{ comp 0} \\
\text{01 } \{[X_1] A\} \\
\text{11 } \{[X_2, X_3] B\}
\end{align*}
\]

In this case with orientation, \( Y = (\text{dyadic add } (1) (X_2)) \) and \( Z = (\text{dyadic max } (X_1) (X_3)) \), since dimension 0 is not an active dimension of \( A \).

### 3.7.3 Dynamic objects

Similar, simple shape-inferences can be applied bottom-up to every node of the F-code tree, and there will then be an F-expression denoting each extent of every operation in the F-code tree. There is a problem, however, when direct references are made to the F-code tree, for an extent expression. Following from example 4, above:

\[
\begin{align*}
\{[X] \text{ ramp } (\square B) \\
\text{(\square E)} \\
\text{(\square S)}
\end{align*}
\]

The shape rule is: \( d_0 = \left[ \frac{d^i - d^f}{c^r} \right] + 1 \). Then \( X \) is the F-expression:

\[
\begin{align*}
\text{(dyadic add} \\
\text{(monadic abs} \\
\text{(dyadic idiv} \\
\text{(dyadic sub B E) } S \\
\text{)} \\
\text{)} \\
\text{(const 1)}
\end{align*}
\]

This is shown diagrammatically in figure 3.4. \( B, E \) and \( S \) can be arbitrary F-expressions, which may include create, or call, or anything at all. Clearly the shape expression for \( X \) will enclose these arbitrary expressions. This is not only inefficient when one calculates the extent of the ramp, but it also possibly incorrect, since F-code is not a referentially transparent language (there may even be assignments in \( B, E \) or \( S \)).

An extent expression will only reference part of the original tree which returns an integer, scalar value. Considering first of all, how the re-evaluation of \( B, E \) or \( S \) can be stopped; or any reference to the original F-tree from an extent expression, the following rewrite to the original tree is made, at the point of the reference:

\[
\begin{align*}
\text{(\square Q)}
\end{align*}
\]
The extent of the RAMP is:

(dyadic add
  (monadic abs
    (dyadic idiv
      (dyadic sub E B)
      S
    )
  )
  (const 1)
 )

... and hence we see that the F-expression representing the extent of the RAMP takes B, E and S as common subexpressions. If B, E and S are simply constants, this is no problem at all; otherwise for those which are not constants, a rewrite is required.

Figure 3.4: Shape expression created for ramp

becomes:

([ ] comma right
  ([ ] assign
    ([ ] var name _x0 )
    ( [ ] Q )
  )
  ([ ] var value _x0 )
 )

Which evaluates Q and leaves behind a side-effect in _x0. The original reference to Q was to evaluate the shape, and then the reference becomes merely

(var value _x0)

Considering the ramp again, where B is a dynamic value, the original tree becomes:

([X] ramp
  ( [ ] comma right
    ( [ ] assign
      ( [ ] var name _x1 )
      ( [ ] B )
    )
    (var value _x1 )
  )
  ( [ ] E )
  ( [ ] S )
)

and X is the F-expression:
CHAPTER 3. THE FRONT END OF AN F-CODE COMPILER, AND INference

```
(dyadic add
  (monadic abs
    (dyadic idiv
      (dyadic sub E (var value _x1))
      S
    )
  )
  (const 1)
)
```

The effect of this rewrite is shown in figure 3.5. The compiler creates an internal variable: 
_x1, _x2, etc. which are global integer variables. One global variable is generated in every instance 
when a reference is made to the original tree by an extent tree: one global variable is created 
for every textual occurrence which needs them.

At the current time, _x1, etc. are global variables which is adequate for a sequential imple­
mentation of F-code — and a version which does not permit recursion. In order to provide a 
version of F-code which permits multiple instances of functions, the scope of _x1, etc. needs to be 
limited to the function. Thereby multiple instances of functions each generate their own ‘global’ 
variables. Similarly for recursive functions, the scope of ‘global’ variables needs to be limited to 
a function body which can be invoked. This will require rewrites to the tree which scope these 
variables at the top of functions/procedures. More on this subject is given in section 3.9.

The introduction of global variable _x1 in the above example decouples expression B from 
the extent expression for X; thus (assuming the value for _x1 is assigned first) the extent tree
3.7. SHAPE INFERENCE

There are three reasons for doing this:

1. B in the previous example may be side-effecting. Which means that executing it more than once may not produce the same result. It thus, simply has to be done.

2. B need not now be executed more than once, when an extent is needed.

3. Decoupling is necessary, such that scope is not violated. An example of this is given on page 84.

If one examines the ordering of the tree after this rewrite has taken place, one observes that global variables _x1, etc. are always read after they are written. The sequentialization described in chapter 4 (section 4.5), due to the instance of a comma makes sure of this.

The partitioning of the dynamically shaped ramp into lazy environments is given in figure 3.6. The assignment environment takes place first. Then, the ramp environment takes place (which may liberally reference _x1 to evaluate the ramp's extent). This example is true for any dynamically shaped tree: the assignment to global variables for holding extents always takes place before they are referenced. The evaluation order is part of the semantics of F-code.
3.7.4 Calls

There is a second case, in a scalar implementation where an extent expression should be made to depend on a variable. This is when an object is returned from a call. It is impossible to know the size of an object returned from a call, unless it is a scalar, without run-time compilation; all one does know is the type, rank and sort. These are attributes of an instance of call. See page 231.

The solution to this is straightforward — to save the size of an object on returning from a function in global variables, and to restore them after the return from call. These global variables could instead be temporary registers, but one needs to make this transformation machine-independent where one can assume no real registers at all. Using global variables is only valid for a sequential implementation of F-code, which does not allow recursion. This is a restriction of the current implementation. Section 3.9 shows how this restriction may be removed.

F-code subroutines

F-code subroutines are identified in the following manner:

- The entire F-tree is a subroutine, returning some object to the environment.
- There are also explicit subroutines defined by the const function of F-code, which defines functions, as was described in chapter 2.

As mentioned above, the rewrite to the subroutine must save all of its extents into global variables. These global variables must have the same name in every subroutine. They are called _c1, _c2, ... etc. _c1 is always used to hold the extent of the lowest dimension, and _c2 the next and so on.

The rewrite to the old subroutine

( [S1,S2] A )

produces the following:

( [_c1, _c2] comma left
  ( [S1, S2] comma left
    ( [S1, S2] A )
    ( □ assign
      ( □ var name _c1 )
      ( □ S1 )
    )
  )
  ( □ assign
    ( □ var name _c2 )
    ( □ S2 )
  )
)

That is it saves the extents in _c1 and _c2. This is an example only for a two-dimensional result; the general adaptation for more or less dimensions should be obvious.
F-code calls

A call to a subroutine needs to use these shape expressions to give the extents of the result of the call. The shape inference for a call depends on the call size variables. In order to do so, the call-size variables must be read and rewritten into more general global variables of the form \(_xN,\ldots\) (See section 3.7.3).

\(([_c1,\ldots,c2] \text{call A <sort> <type> 2})\)

becomes:

\(([_x4,\ldots,x5] \text{comma left})
\hspace{1em} (_[c1,\ldots,c2] \text{comma left})
\hspace{1em} (_[c1,\ldots,c2] \text{call A <sort <type> 2})
\hspace{1em} (\text{assign})
\hspace{1em} (\text{var name }_x4 )
\hspace{1em} (\text{var value }_c1 )
\hspace{1em} )
\hspace{1em} )
\hspace{1em} (\text{assign})
\hspace{1em} (\text{var name }_x5 )
\hspace{1em} (\text{var value }_c2 )
\hspace{1em} )
\hspace{1em} )

The shape of the call is inferred to be \([_x4,\ldots,x5]\).

Call size variables implemented in this way are adequate for a sequential implementation of F-code which may include recursion. How the same effect (passing the size of the result of a call back to an environment) may be achieved in a parallel setting is describe in section 3.9.

3.7.5 Channels

There is a third (and final) case where the shapes of objects are not known at compile time. This is the case for channels. The F-function channel is defined on page 228; the F-function get is defined on page 229. Almost identically to call, the result of get from a channel, has known type, rank and sort, but never shape (unless it is scalar). The solution to this, is just to create a temporary variable for the shape:

\((\text{var value }_x3 )\)

This becomes an expression for a particular dimension. It is left to the code generator for get to take the real value for the extent from fields in the channel (however it is to be implemented) and to assign it to this temporary variable. Nothing more can be done at this stage.
3.8 Complete examples of shape inference

The transformation for shape inference is quite a complex one. In order to demonstrate this last section there now appear a number of increasingly more complex examples.

3.8.1 Example 1

(dyadic max
  (const "hello")
  (const "fred")
)

becomes, first of all, by inferring the shapes:

([4] dyadic max
  ([5] const "hello")
  ([4] const "fred")
)

which is subsequently rewritten, since this is a subroutine (section 3.7.4) to:

([_c1] comma left
  ([4] dyadic max
    ([5] const "hello")
    ([4] const "fred")
  )
  (□ assign
    (□ var name _c1)
    (□ const 4)
  )
)

3.8.2 Example 2

(create A
  (comma right
    (assign
      (var name A)
      (ramp
        (const 1)
        (const 10)
        (const 1)
      )
    )
    (reduce max
      (var value A)
    )
  )
  integer
  (const 5)
)

becomes, first of all by inferring the shapes:
3.8. COMPLETE EXAMPLES OF SHAPE INFERENCE

This is a subroutine which, since it returns a scalar value which is the max of the first 5 values of the ramp, needs no further re-write.

3.8.3 Example 3

(become, first of all by inferring the shapes:

\[
\begin{align*}
& ([x_0] create B \\
& \quad ([x_0] comma right \\
& \quad \quad (assign \\
& \quad \quad \quad (var name B) \\
& \quad \quad \quad (...) \\
& \quad ) \\
& \quad create A \\
& \quad \quad (dyadic max \\
& \quad \quad \quad (var value A) \\
& \quad \quad \quad 0 \ (const 5) \\
& \quad \quad integer \\
& \quad \quad (var value B) \\
& \quad ) \\
& \quad integer \\
& )
\end{align*}
\]
Then function this undergoes a subsequent rewrite which will write \( x_0 \) into \( c_1 \). This rewrite is omitted, but is the same as those in section 3.7.4. The inner create which creates variable \( A \), creates a vector of integers of length \( B \). \( B \) has been previously assigned the value of the expression \ldots

In order to know the shape of the entire subroutine, \( B \) is written into a global register \( x_0 \). This is because the value of \( B \) is only valid at the point of the (create \( A \ldots \)). Inside the scope of \( A, B \) may be assigned another value. This assignment into \( x_0 \) first of all overcomes the lack of referential transparency, and secondly because \( x_0 \) is a global variable it is not dependent on any scope. Consider the following specious inference for the shape of the original expression:

\[
\begin{align*}
([B] &\hspace{1em} \text{create} \hspace{1em} B \\
&\hspace{1em} ([B] \hspace{1em} \text{comma} \hspace{1em} \text{right} \\
&\hspace{1em} ([B] \hspace{1em} \text{assign} \\
&\hspace{1em} ([B] \hspace{1em} \text{var} \hspace{1em} \text{name} \hspace{1em} \text{B} \\
&\hspace{1em} ([B] \hspace{1em} \text{...}) \\
&\hspace{1em} ) \\
&\hspace{1em} ([B] \hspace{1em} \text{create} \hspace{1em} A \\
&\hspace{1em} ([B] \hspace{1em} \text{dyadic} \hspace{1em} \text{max} \\
&\hspace{1em} ([B] \hspace{1em} \text{var} \hspace{1em} \text{value} \hspace{1em} \text{A} \\
&\hspace{1em} 0 \hspace{1em} ([B] \hspace{1em} \text{const} \hspace{1em} 5) \\
&\hspace{1em} ) \\
&\hspace{1em} \text{integer} \\
&\hspace{1em} ([B] \hspace{1em} \text{var} \hspace{1em} \text{value} \hspace{1em} \text{B}) \\
&\hspace{1em} ) \\
&\hspace{1em} \text{integer} \\
\end{align*}
\]

This fails for two reasons:

1. It is liable to side-effects. \( B \) may be over-written in the scope of identifier \( A \).

2. The extent of (create \( B \ldots \)) is supposedly \( B \), however this is actually outside the scope of identifier \( B! \). This is the third reason for the introduction of decoupling variables, from page 79.
3.9 Permitting multiple instances of functions

This is not implemented in the current version of the compiler because the current compiler does not include support for recursive function calls, and since it is for a strictly sequential machine, it does not permit multiple function invocations in this way either.

A function, in the high-level sense, may be identified in F-code as the entire F-tree, or as an F-subtree contained within the const function of F-code which defines functions.

3.9.1 Temporary extent variables

The global variables, _x1 and so on above, must get declared at the edges of functions, and are 'global' to these function bodies. Since one variable name is created for each textual occurrence which needs one, there cannot possibly be interference within a function (or indeed within a program). To permit multiple instances of functions, one instance of a 'global' variable must be created for each invocation of a function. This is simple to achieve in F-code.

If the F-subroutine is:

```
(const
  (A)
)
```

and inside this subroutine, the 'global' variables _x10, _x11, _x12 are required, for instance. The subroutine is rewritten to declare _x10,..._x12 formally:

```
(const
  (create _x10
    (create _x11
      (create _x12
        (A)
        integer
      )
      integer
    )
    integer
  )
)
```

3.9.2 Call size variables

The implementation of call size variables given previously is adequate for a sequential implementation, but not for a parallel implementation which may invoke many calls at any one time. This is because the use of global call size variables will be prone to interference.

What needs to be implemented instead, is for an instance of call to pass a pointer to an area of memory in which the call size is to be written.

This could require the following change to the F-code definition which is only used inside the compiler, and therefore need not be added to the external definition of F-code. This seems to be the only incongruity between the internal and external definitions of F-code.
call is re-defined inside the compiler to have the following syntax:

\[
\text{CALL} ::= (\text{call EXPR <type> <sort> <number> \{CALL\_SIZE\_VARS\}})
\]

Where \text{CALL\_SIZE\_VARS} is a list of identifiers in which to return extents in. Unfortunately, the implementation of this cannot be in terms of an F-code rewrite because it cannot be represented in F-code: the invocation of a call must pass an environment to its destination function (an environment which contains addresses of where the extents are to be saved). There is a trick, however, if one rewrites the function body as usual, including the call size variables \_c1, etc.:

The rewrite to an old subroutine

\[
([S1,S2] A)
\]

becomes as usual:

\[
([_c1,_c2] \text{comma left}

([S1,S2] \text{comma left}

([S1,S2] A)

(\text{assign}

(\text{var name } _c1)

(S1)

)

)

(\text{assign}

(\text{var name } _c2)

(S2)

)

)

The function call to such a function may be

\[
(\text{call } B \text{ value integer 2 } _x4 _x5)
\]

Which calls via B, returning the shape in \(<_x4,_x5>\). These are passed as pointers to the function in some data memory inside the scope of the function. Writes to \_c1, and \_c2 write indirectly through these pointers to \_x4, and \_x5 directly. The implementation requires these special call size variables to be used as indirections:

\[
(\text{var name } _c1)
\]

becomes the equivalent of taking the value of \_c1. The invocation of a function requires the creation of as many call size (scalar) variables as there are dimensions to the result.

The implementation of call size variables is shown diagrammatically in figure 3.7. This diagram invokes two instances of a function, each of which has local call size variables. Writes to call size variables, identifiable because of their lexical strings (\_c1...), become indirect writes to temporary variables within the calling function. In this way, multiple instances of functions are easily attainable.
3.9. PERMITTING MULTIPLE INSTANCES OF FUNCTIONS

Calling function

\_c1, \_c2 are local to each
invocation of a function,
and contain the addresses
of \_<x1, x2>_ and \_<x4, x5>_ to return the extents in.

Figure 3.7: Permitting parallel function invocation

The order of making a function call is thus:

1. Create call size variables for a function. (As many as there are dimensions of the result.)

2. Write to the call size variables the addresses of the extent variables in the calling function which are to receive the shape of the result.

3. Invoke the function. Within the function, writes to call size variables are isolated and in fact write to the address given in the call size variable — write to an extent variable in the calling function.

4. Return the data-parallel result of a function, and destroy the call size variables. The shape of the result will be already be in extent variables _x1... and so on.
3.10 Improving congruence by optimization

One of the fundamental uses of F-code is for data concurrent optimization. F-code is a language which has data-parallel semantics, and importantly for optimization purposes, operational semantics. It is not ad hoc in any way at all. When it is used for reflexive optimization, it does not forfeit simplicity and replace it with a representational explosion.

There are two types of optimization: those which are architecture-neutral and those which are not. Optimization on the F-code representation of an algorithm, or on the algorithm itself, may be architecture-neutral and done at the F-code level. Those which are involved in specific aspects of an implementation are architecture-specific — such as optimizations which make use of a particular hardware feature.

F-code is defined rigorously in terms of data-parallel algebra, a definition of which is given in the previous chapter, and more fully in [BMS92]. The algebra becomes a useful tool for depicting and specifying reflexive optimization transformations. F-code can be optimized firstly without any identification with real machinery at all. These optimizations are not those for pipelining yet, or for inter-processor communication or load-balancing; merely, this is the representation of a computation which may require fewer instructions, or processors, or whatever one intellectually chooses to be a unit of computation, without basis in physical reality at all. There is a very rigid scope scheme in F-code, such that optimization can occur within a framework which is referentially transparent, although F-code itself when viewed as a whole is not.

F-code may be treated, at its front-end, like a black box: a set of data-parallel optimizations; a set which may be increased over time. Targeting compilers need not know anything of the frippery which occurs inside, merely the interfaces.

F-code is nothing more than a parse tree for a data-parallel language; — as such, it is compatible with scalar parse tree optimizations [ASU86], such as common sub-expressions. F-code programs may be littered with data-parallel common sub-expressions.

Since F-code is a data-parallel PSP, there are also geometric operations; F-code is a very powerful denotation for data parallel optimizations. It is possible to reason about — in terms of data-parallel algebra — what happens when algorithms are combined. This provides a large number of legitimate optimizations. Parallelizing compilers, have previously considered standard optimizations like loop fusion. In F-code, it is possible to reason about geometric transformations too, but much more generally than forall loops (which are only able to design 'rectangular' iteration spaces). F-code also has compose (page 225), which is similar in a way to loop-fusion; and many other geometric operations.

In order to do data-parallel optimization, one would like to be able to perform rewrites on an F-code tree, and guarantee that these rewrites produce a tree which is functionally equivalent (and hopefully more efficient). In fact this section is most appropriate for the implementation of a compiler for a distributed machine; and these particular optimizations are not necessary...
for a scalar machine — since the inference of shape expressions, and the implementation of lazy evaluation will make sure that a scalar implementation only evaluates (eagerly) those parts of a computation which are necessary. Other optimizations such as loop fusion are as relevant for a sequential implementation of F-code as for a distributed one; perhaps more relevant.

First of all, what can a data-parallel optimization be? Consider these two variations of a program (which are functionally identical): (see page 223 for definition of slice, and page 222 for definition of sect)

1. (slice 1
   (sect 0
     ...
     (const 2)
     (ramp
       (const 1)
       (ramp
         (const 1)
         (const 3)
         (const 1))
       (const 1))
     (const 1))
   (const 2))
2. (sect 0
   (slice 1
     ...
     (ramp
       (const 1))
     (sect 1)
     (sect 3)
     (const 1))
   (const 2))

Both of these programs are represented in figure 3.8.

The first of these programs should logically do the sect first, the second do a slice first. In a naïve implementation, it would seem that the first alternative would be the most efficient, because it results in a smaller shaded area at the intermediate stage, and hence it requires less memory references. If sect were a hardware operation and slice not, this would indeed be true. If one considers only the indices which are used in performing this operation, the following is true: If we have an index i for dimension 0, and an index j for dimension 1, i is always fixed at the value 2, by the sect operation, and j always takes the series 1, 2, 3 which is fixed by the slice operation. This does not depend on the order between sect and slice. In a similar way, it is clear to say that a referentially transparent section of F-code will regardless of the ordering always be evaluated in the most selective way, if the machine itself is general purpose and orthogonal enough. It is these fundamental incongruities between a machine and a data-parallel language (machines are not general purpose and lack orthogonality) which makes the process of compiling to a parallel machine such a hideous task.
It is the most selective way, but it might not necessarily be the most efficient implementation. An F-code program must be rewritten in order to take advantage of, or to avoid parallel hardware features and restrictions respectively. A very simple kind of data-parallel optimization may be, therefore, the kind which moves \texttt{sect} down the tree if it is a hardware operation. This is an architecture-dependent optimization. These are the simplest types of optimization, which just depend upon rewrites which are short-range (applied to two functions which are adjoining), and they are shown for examples only; no analysis of their improvement in ‘efficiency’ (however one adjudges that?) is given. This is a good area of research for the future.

3.10.1 An exercise in data-parallel algebra

In this section is presented a number of local transformations on F-code programs which given \( f \circ g \), give some \( g' \circ f' \) — that is \( f \) is propagated down the tree and executed before \( g \) since it is nearer the leaves, and \( f \mapsto f' \) and \( g \mapsto g' \) are side-effects of the rewrite.

In order to perform some of the rewrites, a function \( b_j \) is needed which provides the extent of a masked operand along a particular dimension. \( b_j \) gives the extent along dimension \( j \) of operand \( a \) already projected onto its corresponding mask \( m^a \).

In the previous sections of the chapter it is shown how this extent is inferred from the original F-code source text, and that the extent may either be an integer constant, if extents are statically known at compile time, or it may be a function evaluating to an integer constant.

Propagation of Sect

These are a few examples of optimization rewrites given for \texttt{sect}, the idea of which is to propagate the \texttt{sect} further down the tree towards the leaves. One would like to propagate \texttt{sect} for example in instances where \texttt{sect} is an available hardware feature.

Some rewrites are very straightforward, such that

\[
(SECT \ n \ (\ldots) \ i) \mapsto (\alpha \ (SECT \ n \ (\ldots) \ i))
\]

where \( \alpha \in \{\text{HOLD, TEMPLATE, CREATE, MONADIC, PART, CHANNEL, COMMA}\} \). These are functions which have a primary operand which is returned as the evaluation of the function without modification. It thus does not matter whether \texttt{sect} is applied before or after (inside or outside) one of these functions.

Other rewrites are more complicated. One defines the application of \( (SECT \ n \ ... \ i) \) to an operand expression \( x \) with an orientation mask, \( m^x \). Here \( n \) is the dimension to which the \texttt{SECT} applies and \( i \) an expression which returns an integer value for the fixed index. This results in an operand expression \( \tilde{x} \) and a modified mask \( \tilde{m}^x \), where

\[
\tilde{x} = \begin{cases} 
(SECT \ U(n, m) \times i), & m^x_n = 1 \\
x, & \text{otherwise}
\end{cases}
\]

where \( U(k, m) \) is the number of mask bits \( k_j = 1 \) with \( j < k \).
3.10. IMPROVING CONGRUENCE BY OPTIMIZATION

\[ m^*_k = \begin{cases} 
  m^*_k, & k < n \\
  m^*_k, & n \leq k \leq r(m^*) - 2 
\end{cases} \]

\( r(m) \) is the length of mask \( m \); the length of the result mask is then \( r(m^*) - 2 \).

It is then quite easy to formulate more complicated rewrites for \( \text{SECT} \):

\[
(\text{SECT} n \ (\text{DYADIC} \bigoplus m^l \ l^m r) \ i) \mapsto (\text{DYADIC} \bigoplus m^l \ l^m r)
\]

\[
(\text{SECT} n \ (\text{CHOICE} \ m^t \ s \ m^t \ t \ m^t f) \ i) \mapsto (\text{CHOICE} \ m^t \ s \ m^t \ t \ m^t f)
\]

\[
(\text{SECT} n \ (\text{DISPLACE} \ k \ m^p \ p \ m^t s) \ i) \mapsto (\text{DISPLACE} \ k \ m^p \ p \ m^t s)
\]

\[
(\text{SECT} n \ (\text{DISTANCE} \ k \ m^l \ l^m r) \ i) \mapsto (\text{DISTANCE} \ k \ m^l \ l^m r)
\]

\[
(\text{SECT} n \ (\text{COMP} \ k \ m^l \ l^m r) \ i) \mapsto \begin{cases} 
  (\text{COMP} \ k \ m^l \ l^m \ r), & k \neq n \\
  (\text{SECT} n \ [m^l \ i] \ i), & k = n, i < b^l_n \\
  (\text{SECT} n \ [m^r \ r] \ (\text{DYADIC} \ \text{SUBS} \ i \ b^l_n)), & k = n, i \geq b^l_n
\end{cases}
\]

where \([m^x z] \) means an expression with mask. Unfortunately this is counter to external F-code syntax, and is a result arrived at at some intermediate stage of a rewrite. This may be rewritten further, and is valid subject to further restrictions:

\[
(\text{SECT} n \ [m^x z] \ x) \mapsto \begin{cases} 
  x, & m^x_n = 0, m^y_n = 1, y \neq n \\
  (\text{SECT} n \ x \ z), & \forall y, m^y_n = 1
\end{cases}
\]

It cannot be re-written otherwise (if there is more than one zero in the mask, or if there is one zero in the mask and it does not correspond to the \( n \) dimension specified for the \( \text{SECT} \).

\[
(\text{SECT} n \ (\text{REDUCE} \bigoplus m^a \ a) \ i) \mapsto (\text{REDUCE} \bigoplus m^a \ a), \ m^a_n = 0
\]

\[
(\text{SECT} n \ (\text{TRANSFORM} \ s \ \{m^t \}...) \ i) \mapsto (\text{TRANSFORM} \ s \ \{m^t \ t\}...) \]

where \( \{m^x \}... \) means optional and repeated, and is part of the syntax of P-code.

\[
(\text{SECT} n \ (\text{POL} \ c \ \{m^v \ v\}...) \ i) \mapsto (\text{POL} \ c \ \{m^v \ v\}...) 
\]

\[
(\text{SECT} n \ (\text{SLICE} \ k \ a \ v) \ i) \mapsto \begin{cases} 
  (\text{SLICE} \ k \ (\text{SECT} n \ a \ i) \ e), & k \neq n \\
  (\text{SECT} n \ a \ (\text{SECT} 0 \ e \ i)), & k = n
\end{cases}
\]

where

\[ k = \begin{cases} 
  k, & k < n \\
  k - 1, & k > n
\end{cases} \]

**Propagation of Slice**

These are a few examples of optimization rewrites given for slice, the idea of which is to propagate the slice further down the tree towards the leaves. One would like to propagate slice towards the leaves, but not below \( \text{sect} \) since \( \text{sect} \) is generally more selective. Thus there is no rewrite given for \( \text{SLICE} \circ \text{SECT} \), although there was previously for \( \text{SECT} \circ \text{SLICE} \).

Again, some rewrites are very straightforward, such that

\[
(\text{SLICE} n \ (\alpha (...) \ i) \mapsto (\alpha \ (\text{SLICE} n (...) \ i))
\]
where $\alpha \in \{\text{HOLD, TEMPLATE, CREATE, MONADIC, PART, CHANNEL, COMMA}\}$

These are functions which have a primary operand which is returned as the evaluation of the function without modification. It thus does not matter whether $\text{slice}$ is applied before or after (inside or outside) one of these functions.

Again, other rewrites are more complicated. One defines the application of $(\text{SLICE } n \ldots i)$ to an operand expression $x$ with an orientation mask, $m^x$. Here $n$ is the dimension to which the $\text{SLICE}$ applies and $i$ an expression which returns an integer vector value. The result is an operand expression $\hat{x}$ and a modified mask $\hat{m}^x$, where

$$\hat{x} = \begin{cases} (\text{SLICE} U(n, m) x i), & m^x_n = 1 \\ (\text{REPL} n x b^x), & m^x_n = 0 \end{cases}$$

where $U(k, m)$ is the number of mask bits $k_j = 1$ with $j < k$.

$$\hat{m}^x = \begin{cases} m^x, & m^x_n = 1 \\ \emptyset, & m^x_n = 0 \end{cases}$$

where $\emptyset$ signifies the default mask of all ones.

It is then quite easy to formulate more complicated rewrites for $\text{sect}$:

$$(\text{SLICE } n (\text{DYADIC} \odot m^l \odot m^r) i) \mapsto (\text{DYADIC} \odot \hat{m}^l \odot \hat{m}^r)$$

$$(\text{SLICE } n (\text{CHOICE} m^s \odot m^t \odot m^f) i) \mapsto (\text{CHOICE} \hat{m}^s \odot \hat{m}^t \odot \hat{m}^f)$$

$$(\text{SLICE } n (\text{DISPLACE} k m^p \odot m^s) i) \mapsto (\text{DISPLACE} k \hat{m}^p \odot \hat{m}^s)$$

$$(\text{SLICE } n (\text{REDUCE} \odot m^a) i) \mapsto (\text{REDUCE} \odot \hat{m}^a), m^a_n = 0$$

$$(\text{SLICE } n (\text{TRANSFORM} s \{m^l i\}...) i) \mapsto (\text{TRANSFORM} s \{\hat{m}^l i\}...)$$

where $\{m^s x\}...$ means optional and repeated

$$(\text{SLICE } n (\text{POL c} \{m^v u\}...) i) \mapsto (\text{POL c} \{\hat{m}^v u\}...)$$

$$(\text{SLICE } n (\text{COMP} k m^l \odot m^r) i) \mapsto (\text{COMP} k \hat{m}^l \odot \hat{m}^r), k \neq n$$

Propagation of Transp

Finally, here are a few examples of optimization rewrites given for $\text{transp}$. $\text{transp}$ is only an operation on indices. In the sequential implementation it takes no time to 'compute a transp'. This may not be the case in a distributed implementation. $\text{transp}$ operations can be propagated to the leaves of the $F$-tree.

Again, some rewrites are straightforward, such that

$$(\text{TRANS} n (\alpha (...) \mapsto (\alpha (\text{TRANS} n (...)))$$
3.10. IMPROVING CONGRUENCE BY OPTIMIZATION

where $\alpha \in \{\text{HOLD, TEMPLATE, CREATE, MONADIC, PART, CHANNEL, COMMA}\}$

First see the definition of transp on page 222, since the definition for the mapping

$$T : (a, N) \rightarrow b$$

is used again here. For other rewrites, being more complicated, one defines the application of $(\text{TRANS} n \ldots)$ to an operand expression $x$ with an orientation mask, $m^x$. Here $n$ is the dimension of the TRANS. This rewrite results in an operand expression $\widehat{x}$ and a modified mask $\widehat{m}^x$.

First one defines an identity vector $i^a$ which is such that

$$i^a_i = i, \quad 0 \leq i < r(e^a)$$

$$\widehat{m}^x_i = \begin{cases} 1, & i \in p(T(i^a, n)) \text{ and } 0 \leq i < r(e^a) \\ 0, & \text{otherwise} \end{cases}$$

$$\widehat{x}^a = \begin{cases} (\text{TRANS} k x^a), & k \neq \infty \\ x, & \text{otherwise} \end{cases}$$

where

$$k = h(p(T(i^a, n)), n)$$

and

$$h(y, r) = \begin{cases} \infty, & T'(\epsilon(y), r) = \infty \\ i, & p(T'(\epsilon(y), r))_{i} = r \end{cases}$$

where $T'$ is the inverse function of $T$, such that

$$T'(T(x, r), r) \equiv x$$

the definition of which is omitted.

$$(\text{TRANS} n (\text{DYADIC} \circ m l m^r r)) \rightarrow (\text{DYADIC} \circ \widehat{m} l \widehat{m}^r \widehat{r})$$

$$(\text{TRANS} n (\text{CHOICE} m^s m^t n m^f f)) \rightarrow (\text{CHOICE} \widehat{m}^s \widehat{m}^t \widehat{m} l \widehat{m}^f \widehat{f})$$

$$(\text{TRANS} n (\text{DISPLACE} k m^p p m^q q)) \rightarrow (\text{DISPLACE} k \widehat{m}^p \widehat{m}^q \widehat{p} \widehat{q})$$

$$(\text{TRANS} n (\text{REDUCE} \circ m a)) \rightarrow (\text{REDUCE} \circ \widehat{m} (\text{TRANS} n' a))$$

where

$$n' = i, E(i^a, \widehat{m}^a)_{i} = n$$

$$(\text{TRANS} n (\text{TRANSFORM} s \{m^t t\} \ldots)) \rightarrow (\text{TRANSFORM} s \{\widehat{m}^t \hat{t}\} \ldots)$$

$$(\text{TRANS} n (\text{POL} c \{m^v u\} \ldots)) \rightarrow (\text{POL} c \{\widehat{m}^v \hat{u}\} \ldots)$$

$$\begin{cases} (\text{TRANS} n' (\text{SECT} k a i)) \rightarrow \\ (\text{SECT} k' (\text{TRANS} n' a) i) \quad \begin{cases} n < k, & n' = n, m' = m - 1 \\ n \geq k, & n' = n + 1, m' = m \end{cases} \\ (\text{SECT} k' a i) \quad \begin{cases} n < k, & k' = k - 1 \\ n \geq k, & k' = k \end{cases} \end{cases}$$

where $x$ is the original expression $(\text{TRANS} n (\text{SECT} k a i))$
3.10.2 Conclusions

Laconically, other optimizations have been described like loop fusion. These are examples of long-range optimizations — by which is meant optimizations which are not limited to two F-code functions which are adjoining (like the ones written above, in section 3.10.1.) These types of optimization are completely different in nature, since they require a pattern matching technique to find them, whereas the optimizations shown previously can be performed mechanically using a recursion through the F-tree. Both types need some metric to prove the gain in efficiency in making use of them; perhaps a rule-based system, or learning algorithm of some form.

This section includes the rudiments of a set of rewrites to improve the efficiency of F-code on a distributed machine. The rewrites are a benefit of the use of an algebra to define F-code in the first place. It is also a result in its own right about the attractiveness of using a data-parallel algebra to specify and perhaps with further research in the area of data-parallel algebra to prove the efficacy of data-parallel optimization rewrites. F-code is the first example (perhaps of many) of a data-parallel portable software platform and tool, or rather means for data-parallel algebraic manipulation.

Of course, no judgement or quantitative analysis has been done on the optimization examples given above; this is because this section is diversionary to the main gist of the thesis and is more particularly suitable for a distributed implementation. The section has only shown that algebraic manipulation is possible, although the results can seem quite difficult to understand — this abstruseness is due to the complexity of the problem of specifying data-parallel computation, and not a problem or limitation of the algebra.

Short-range optimizations have been shown which re-combine two functions which are adjacent in the opposite order. Just the smallest set of legitimate optimizations have been shown; there are potentially a vast number of others. This section has shown however, the tractable use of F-code's algebra to specify manipulations; perhaps it needs to be augmented with some means of evaluating a function's efficiency, to prove the optimization.

3.11 Summary

This chapter showed architecture-independent aspects of compiling F-code.

1. The manner of parsing F-code is given, which uses a table so that the parser is easily modifiable. This part of the compiler is not expected to be rewritten, since it is an architecture-neutral part of the compiler. However, since this is only a prototype compiler, F-code itself may change some amount, in time.

2. F-code programs are type-inferrable, from the minimum amount of type information. Also attractively, there is no need to explicitly place type-coercions. A type-inference stage of compilation can place type-coercions automatically, and the program will be strongly typed. Since F-code is type-inferrable, it proves the presumption that one can
omit unnecessary type declarations in an original F-code source program. This means that the original text can be smaller.

The type-inference algorithm also makes use of the parse table, and thus like the parser this is easily modifiable.

3. **F-code programs are shape-inferrable** at compile-time. Of course this is vital if a program is to be compiled at all, without a complex run-time system. It is possible to infer the shape of every node of the program, even when the program uses dynamic shapes. Where a program is dynamic, the type inference stage of the compiler creates F-code expressions which evaluate at run-time to give the shapes.

4. **Congruence can be improved by high-level optimization.** Although shape inference allows the lazy-implementation of the most selective area of iteration spaces, a set of rewrite rules can be used to make the computation more efficient on a parallel computer. It is out of the time-scale of this thesis to implement these; and is also unnecessary since the aim of this thesis is the concrete implementation of F-code on a sequential scalar machine. In fact, one could possibly give an alternative name for Portable Software Platform in the data-parallel domain and call it a data-parallel algebra tool, since this could be its primary purpose.

These optimization rewrites must preserve the results of type and shape inference. Since these optimizations have yet to be implemented, this is an area of future work.

The optimizations demonstrated are not useful for the current implementation of F-code to an i860. Lazy evaluation makes sure that the least calculation is done in any program: only elemental operations which affect the result of a program are executed. The implementation is as selective as it can possibly be. These optimizations would mainly be used to take advantage of inherent machine instructions for a sequential machine: for example if a machine had a slice function, slice ought to be propagated towards the leaves of a tree. Only scalar operations exist on an i860, and thus the optimizations are not done.

These optimizations are more useful for parallel machines because the order in which computation takes place may radically change the communication pattern of a program.

The next chapter deals with maintaining the sequential dependencies implicit in F-code, for the purposes of compiling F-code to a sequential machines. Most of the ideas in the next chapter are also relevant to the distributed case.
Chapter 4

Sequential aspects of F-code

Figure 4.1: Sequentialization compiler phases

- 4.1 — Introduction to this chapter .......................................................... 98
- 4.2 — Identifying lazy environments ....................................................... 99
- 4.3 — Lazy evaluation across index spaces ............................................. 101
- 4.4 — Indices ......................................................................................... 108
- 4.5 — Sequential ordering of an F-code tree ............................................ 116
- 4.6 — Summary and conclusions ............................................................ 125
- 4.7 — Applying this chapter to parallel machines ................................. 126
4.1 Introduction

This chapter is mainly applicable to an implementation on a scalar, sequential machine. However, section 4.2 is also applicable to a purely distributed implementation, since it has a dual meaning there (this is explained in section 4.7). The diction is in all sections, however, one to discuss an implementation on a scalar, sequential machine, since it is these kind of machines which concern this thesis most.

This particular chapter is not concerned at all with fine-grain parallelism / scheduling. Instead only necessary sequentializations are dealt with here: that is such-and-such has to occur before such-and-such at a coarse level to conform to the semantics of F-code, and no implementational details are considered at all. Scheduling issues are part of the code-generator which is in another chapter. Even those reading this thesis superficially, or implementors of F-code compilers for a distributed machines, should read this section to understand the gist of the chapter and finally read section 4.7 to consolidate this information.

A compiler for a distributed computer should employ a sequential compiler to utilize fine-grain parallelism on single processors: Figure 4.2.

![Diagram](image)

Figure 4.2: Subordination of a sequential compiler in a distributed one

This thesis is most interested in the sequential compiler, since it is a prerequisite for the distributed one. One assumes that the sequential compiler compiles full-specification F-code, but segments of F-code which are limited in some way by the distributed compiler. This is equally applicable if one assumes that in a distributed machine each processor has a copy of
identical segments of code, or if each processor has its own specialized segments of code, perhaps in an inhomogeneous array of processors. This need not be cogitated at this stage.

The two compilers which are used to compile an F-code program to a parallel machine may use the same architecturally independent stages which were described in the previous chapter and then become completely different in all subsequent stages. The distributed compiler calls on the sequential compiler to compile small sections of F-code — perhaps node programs.

The discussion henceforth is limited to the sequential compiler (by which is meant a compiler for a scalar, or a vector processor, pipelined or not — basically a single autonomous processor, perhaps in an array of processors). From a software engineering standpoint, the sequential compiler assumes a single processor machine, has the same alphabet as a full, distributed compiler and is entirely subordinate to the full compiler. Its interfaces are thus — they seem initially to be — the same as the full compiler.

The source graph, obtained by the compiler to this point, has type information, shape information, but has no explicit notion of data-parallel loops. It is thus necessary to find trees upon which the semantics of loops are applied — which are lazy environments. Then it is necessary to examine the order in which lazy environments may be executed in respect to each other. And how these ‘lazy’ environments may be evaluated lazily, and their memory accesses optimized.

### 4.2 Identifying lazy environments

It has previously been described that an F-code program is conceptualized as a number of lazy environments and eager functions. Those eager functions are put, hold, template, create,
coerce, mark, and call. These functions are eager because they have side-effects — which
means that their primary parameter cannot be executed lazily, but must be evaluated as a block
result, in a lazy environment. The order of create is, for example: (1) create object (2) trigger
the evaluation of its primary operand (which can be a lazy environment) (3) destroy object.

The algorithm to identify the top of lazy environments is quite straightforward. The top
of a lazy environment is identified by the fact that it is a node (other than one of the eager
functions above) which is directly below, adjacent to, an eager function, and it is a node
which has a non-scalar result. Secondly a lazy environment may be tagged to be the top of the
entire F-program, if it returns a non-scalar result. There is an assumption of this compiler that
the discarded operand of a comma returns a scalar value only. Otherwise, this algorithm must
be augmented with the fact that a discarded operand of comma which returns a non-scalar value
must be tagged. This assumption holds because, logically, the discarded operand of a comma will
only ever be an assignment, or set of assignments (which, by virtue of F-code semantics, return
a scalar value). This algorithm is called loop filtering.

A node which fulfills the requirements of loop filtering is tagged with a tag called loop, such
as is shown in figure 4.3. The top of the lazy environment describes a node in the tree for which
the code generator is required to generate loops to implement data-parallelism. The leaves of the
lazy environment are the nodes above eager functions, or if there is no intervening eager function,
leaves of the F-tree itself. If all eager functions in the tree are ascribed with a tag, say eager,
then the portion of the tree which is called a lazy environment is from the root which is tagged
with loop, down to nodes just above nodes tagged with eager, or down to leaves of the F-tree.
This tagging for eager need not take place — it is implicit because only certain F-functions are
eager. The loops required to implement data-parallelism are called inner-autonomous loops —
which are generated at the nodes labelled loop by the code generator.

There is a second kind of loop in F-code required to implement the semantics of F-functions,
such as loops required to implement reduce, which do not need tagging in the tree. These are
identifiable by the lexical name of the F-function. These are internal indices, as will be shown on
page 111 for reduce and other functions. These indices (and loops) carry out the semantics for
individual, elemental results of a lazy environment, and are enclosed by the lazy environment.
They can be ignored for purposes of identifying lazy environments.

Let us now give a few examples of finding lazy environments in programs:

( [4] dyadic max           loop
   ( [5] const "hello" )
   ( [4] const "fred" )
)
4.3 Lazy evaluation across index spaces

This chapter is about the identification of lazy environments, and their evaluation, and ordering with respect to each other. This section describes how index spaces are implemented within lazy environments. Shape inference in the previous chapter has created a number of F-expressions for every F-function in the primary tree which give the extents of the index space of the F-function's result. The shape is a vector, matrix, box, etc. to any dimension because of the shape coercion rules of F-code.

A lazy evaluation, once identified (section 4.2), will already have the shape of its index space inferred. Lazy evaluation means that only those elements of the index space which affect the result of a lazy environment are executed.

It will be shown that the shape expressions inferred by shape inference can be used to distribute indices across the primary F-tree, just by following their connectivity. F-functions
which create new binding instances of indices, like \texttt{reduce}, which uses these internal indices also use the shape expressions to distribute their indices.

Figure 4.4 shows a lazy environment which returns a vector result which takes the average along a particular dimension of a matrix, contained in A, whose shape is \langle 5, 10 \rangle. The shape of the result is the same as the first extent of A. The lazy environment needs one index to execute (since it returns a vector). The \texttt{reduce} needs a further internal binding instance of an index to carry out the reduction.

If one goes on to label the indices in this tree — this is easily done so by following the connectivity of the shape expressions —, the result is as in figure 4.5. Here, \texttt{i2} is an internal index generated for the \texttt{reduce}, and \texttt{i1} is an index used to implement the data-parallelism of the lazy environment. The annotation of the program, between square brackets, this time is the indices required to implement the program. This annotation is not an addition to F-code's syntax: the square brackets are just used to show different elements of the same thing: indices are shown in this chapter to be attributes ascribed to the shape-inference expressions, and the annotation shows either the shape or the index which implements the data-parallelism.

This lazy environment can be implemented using the following C program:

```c
for (i1=0; i1<5; i1++) {
    v1 = 0;
    for (i2=0; i2<10; i2++) {
        v1 += A[i1,i2];
    }
    v1 /= 5;
    result[i1] = v1;
}
return result;
```

Now consider a lazy environment which does the equivalent of composing together two index spaces: using the \texttt{comp} function of F-code. An example use of this function, figure 4.6, composes together two additions along the first dimension (dimension 0). The \texttt{comp} has to create a new (internal) index for dimension 0 of the right hand side of the \texttt{comp}. This is shown formally on page 111. The value of this index is calculated from one of the lazy environment's indices as shown in the figure.
4.3. LAZY EVALUATION ACROSS INDEX SPACES

Figure 4.6: Composing index spaces with F-function comp
For the program \( \text{comp } 0 \) (dyadic add \( A \) \( B \)) (dyadic add \( C \) \( D \)), the figure shows, first of all, the shape inference for the F-program where the shapes of \( A,B,C,D \) are \(<X1,X2>, <X3,X4>, <X5,X6>, <X7,X8>\), respectively. The index spaces required to implement the compose for this function are \( i1,i2 \) to evaluate the left hand and \( i3,i2 \) to evaluate the right hand side. These indices are labelled by following the shape trees. \( i3 \) is an internal index for dimension 0 of the right hand side.

This F-program may be implemented by the following C program:

\[
\begin{align*}
e0 &= \min(X1,X3); \\
e1 &= e0+\min(X5,X7); \\
e2 &= \min(\min(X2,X4),\min(X6,X8)); \\
/* The shape of the result is <e1,e2>. e0 holds \( \min(X1,X3) \) */ \\
for (i1=0; i1<e1; i1++) { \\
    for (i2=0; i2<e2; i2++) { \\
        if (i1 > e0) { \\
            /* Do right hand side of compose */ \\
            i3 = i1 - e0; \\
            /* i3,i2 is the index space of this side */ \\
            v1 = (C[i3,i2]+D[i3,i2]); \\
        } \\
        else { \\
            /* Do left hand side of compose */ \\
            /* i1,i2 is the index space of this side */ \\
            v1 = (A[i1,i2]+B[i1,i2]); \\
        } \\
        result[i1,i2] = v1; \\
    } \\
} \\
return v1;
\]

The next example shows that the F-code function diag may be executed with no cost (except how it affects the communication structure of a program) because of labelling the indices in such a way. Figure 4.7 shows a lazy environment which takes the diagonal of its operand, which is a box.

The index space of the result is just \( i1 \), the index space of the operand is \( i1,i1,i1 \). Only the diagonal of the operand is computed: this is the best example of lazy evaluation. If one makes \( A \), from the figure, some F-function like (dyadic add \( B \) \( C \)), whose shape is \(<X1,X2,X3>\) as before, this environment can be computed by the C program:

\[
\begin{align*}
e0 &= \min(X1,X2,X3); \\
for (i1=0; i1<e0; i1++) { \\
    /* The index space of the diag is <i1,i1,i1> */ \\
    result[i1] = B[i1,i1,i1] + C[i1,i1,i1]; \\
} \\
return result;
\]

The diag is only present in terms of what it does to the index space, and is therefore computed with no cost: it does not introduce code to the implementation.
4.3. LAZY EVALUATION ACROSS INDEX SPACES

4.3.1 Optimizing data-parallel loops

This section has so far shown how lazy environments may be implemented by the introduction of loops to carry out data-parallelism. This part of the section describes how such loops can be optimized by the compiler. Loops are positioned at the top of lazy environments. (Whose positions are inferred by loop filtering.) The code generator generates for each node tagged with loop, a lazy environment whose indices form an index space. It generates loops to scan across this index space.

The shape inference procedure of F-code, and the subsequent labelling of indices provides an implementation which is lazy: the most selective computation is carried out.

A lazy environment consists of:

1. A primary rectangular index space, for which a shape has been inferred.
2. A parallel result datum (whose shape is the same as the index space).
3. Index manipulations which do arithmetic operations on indices to provide values for other, internally bound instances of variables: such as sect which creates an index, and fixes it at a value. (Page 111.)
4. Memory references (in terms of indices).
5. Elemental computations (which are not the subject of this section)

The index space (index spaces, when internal internal indices are created) is already as selective as possible, and index manipulations are already at a minimum — without rewriting the F-code tree in any instance. This is because of the implementation of lazy evaluation.

The lazy environment is implemented eagerly, elementally: it is scheduled like any conventional, non-lazy program. The laziness is provided by the selectivity of the index spaces. Which cannot be improved.
There are two sources of optimization for a lazy environment: (1) memory dereferencing is done in terms of indices, such as $A[i_1, i_2, i_3]$, the computation of an address for which is costly because it implies multiplication — such as $A + i_1 + (i_2 \cdot \text{ext}_1) + (i_3 \cdot \text{ext}_1 \cdot \text{ext}_2)$ (2) writing to the result datum need not be done in terms of indices, and a pre/post-incrementing addressing mode may be instituted on a sequential processor. This is often a cause of super-linear speedup for fully distributable operations — i.e. no computation of addresses.

Consider an F-code tree, with is a lazy environment, such as:

```
([e0, e1] dyadic add
  ([e2, e3] var value A)
  ([e4, e5] var value B)
)
```

If indices are defined $i_d$, where $d$ is the active dimension of the result, this can be implemented, storing the result of the operation in a data-parallel variable called result (which is just used to serve its purpose in this example) by

$$\text{result}[i_0, i_1] = (A[i_0, i_1] + B[i_0, i_1]), 0 < i_0 < e_0, 0 < i_1 < e_1$$

where

$$e_0 = \text{min}(e_2, e_4), e_1 = \text{min}(e_3, e_5)$$

If memory is mapped contiguously, as it usually is, this is physically implemented by

$$\text{result}[i_0 + (e_0 \cdot i_1)] = (A[i_0 + (e_2 \cdot i_1]) + B[i_0 + (e_4 \cdot i_1)]), 0 < i_0 < e_0, 0 < i_1 < e_1$$

$e_x$ are arbitrary F-expressions, which evaluate to integer scalar constants, or they may just be immediate integer scalar constants. The multiplications in the above implementation are very costly. In the case where $e_0 = e_2 = e_4$ this can be implemented using a single loop, and one index $i_0$ whose extent is $e_0 \cdot e_1$, (this is a standard technique found in for example [PK87], called loop coalescing).

$$\text{result}[i_0] = A[i_0] + B[i_0], 0 \leq i_0 < (e_0 \cdot e_1)$$

The F-code compiler as yet does not implement loop coalescing, and assumes that loops cannot be coalesced.

In the case where it needs to be implemented by two loops (where loop coalescing should not be used, or in the current implementation where loop coalescing is not implemented) there are a number of multiplications to evaluate multi-indices. Each multiplication can be replaced with a counter, in a sequential implementation. If factor $e_x$ in the above expression is replaced with a counter variable $c_{e_x}$, and since the result is always writing to contiguous memory locations, a post-increment write via a pointer may be used:
4.3. LAZY EVALUATION ACROSS INDEX SPACES

```c
r=result;
for (i1=ce2=ce4=0;i1<e1;i1++) {  
  for (i0=0;i0<e0;i0++) {
    *r++ = A[i0+ce2]+B[i0+ce4];
  }
}
```

Secondly one may search, at compile-time, for common-factors, such that if in the example, $e_2$ is known to be the same as $e_4$, by expression comparison, it may be further reduced to:

```c
r=result;
for (i1=ce2=0;i1<e1;i1++) {  
  for (i0=0;i0<e0;i0++) {
    *r++ = A[i0+ce2]+B[i0+ce2];
  }
}
```

Ordinarily, $\ldots = X[y] \ldots$ is a hardware operation, as is $\{ *X++ \ldots \}$

These optimizations finds their way into the code generation stage of the compiler: the T-code node loop (which will be shown in chapter 5, section 5.2.7). The counters for loop are taken out into a `cmb` block (which defines factors).

Finally, there are loop invariants to contend with. In this example $e_0$, $e_1$ and $e_2$ are all invariant, and can be taken outside both loops if they are expressions and not simple constants. $e_0$, $e_1$, $e_2$ are the evaluation of shape expressions (created by shape inference). They are loop invariants with respect to the loop that implements data-parallelism:

A real example of this factorization technique is given:

```plaintext
([[e_0, e_1] dyadic add  
  ([e_2, e_3] dyadic mul  
    ([e_4, e_5] var value A)  
    ([e_6, e_7] var value B)  
  )  
  )  
01 ([e_8] var value C) )
```

```plaintext
e_2 ::= (dyadic min e_4 e_6)  
e_3 ::= (dyadic min e_5 e_7)  
e_0 ::= e_2  
e_1 ::= (dyadic min e_3 e_8)
```

This can be implemented:

```
result[i_0, i_1] = (A[i_0, i_1] * B[i_0, i_1]) + C[i_1], 0 \leq i_0 < e_0, 0 \leq i_1 < e_1
```

in terms of multiplications:

```
result[i_0 + (e_0 * i_1)] = (A[i_0 + (e_4 * i_1)] * B[i_0 + (e_6 * i_1)]) + C[i_1], 0 \leq i_0 < e_0, 0 \leq i_1 < e_1
```

considering factors of $i_1$ in the above example, if $e_0 = e_4 = e_6$, this can be implemented using a single loop, which can not be implemented efficiently due to the division or modulus (which
is more inefficient than the control structure for a loop, depending on rank):

\[ \text{result}[i_0] = (A[i_0] \ast B[i_0]) + (C[i_0]), 0 \leq i_0 < (e_0 \ast e_1) \]

Instead, the pair of loops is implemented:

```c
for (i1=c4=0;i1<e1;i1++,c4+=e4,c6+=e6) {
    for (i0=0;i0<e0;i0++) {
        *r++=A[i0+c4]*B[i0+c6]+C[i1];
    }
}
```

And e0, e1, e4 and e6 should be pre-computed, if they are not constants, since they are loop-invariants.

If e4=e6, this can be reduced, automatically to

```c
r=result;
for (i1=c4=0;i1<e1;i1++,c4+=e4) {
    for (i0=0;i0<e0;i0++) {
        *r++=A[i0+c4]*B[i0+c4]+C[i1];
    }
}
```

Again, e0, e1 and e4 should be pre-computed, if they are not constants, since they are also loop-invariants. This pre-computation is done quite simply because shape-expressions should be evaluated before the lazy environment (loop) to which they apply, anyway. The reduction is done by the code generator at the time it generates loops: it creates only counter variables whose increments are unique, thereby limiting the number of indices to the minimum necessary.

### 4.4 Indices

Before considering the sequential ordering between lazy environments, it is useful to describe the implementation of indices; and the algorithm which ornaments F-code trees with indices. In a sequential implementation, indices are needed to evaluate what are conceptually data-parallel operations. These indices are implicit only in an F-code representation since F-code is a PSP for data-parallel programming: no assumption is made to the manner of implementation: the implementor must infer the positions of indices.

The indices are, at this stage, represented by somewhat arbitrary labels; arbitrary in all senses except that indices may be discerned between — because all instances of a particular index have a unique label. The label corresponds to a register or variable in the machine-level implementation which carries out the index's purpose — which is to provide an index space inside a lazy environment.

A primitive object is indexed with a multi-index, which is a vector of indices. A standard operation like dyadic is defined to have the contents (page 221):

\[ c_k = c^l_{p(k)} \odot c^r_{p(k)}, \ k \prec d, \]
and extents:
\[ d = \min(e^l, e^r) \]

In the previous chapter, it was shown how this denotation for the shape of the result was used to construct the shape of every node in the F-tree. This algebraic expression is a relation between shapes. Indices used in a program are to be enumerated in this section. It will be shown that indices can be associated with particular F-functions by percolating them down the shape-expression trees. The shape expression trees are a guide to which F-function uses which index. (This is shown in subsection 4.4.3).

4.4.1 Notation for indices

There are in effect two kinds of index: those which apply to the lazy environment's main loops (which create the index space required to compute the result of a lazy environment); and indices which are generated inside the lazy environment called internal indices (internal binding occurrences of index variables.) The values of internal indices may be computed from those of other indices (such as is the case for the internal index created for F-function slice), or may be used to implement internal index spaces for F-functions like reduce.

1. A label is the literal name of an index. It does not matter how it may be represented, except individual labels are unique and can be compared for equality. They may be scalar integers, for example. They have been shown and used previously in the form of \( i_0, i_1, \) etc.

2. For an operation in the F-code tree which creates some result, the index vector \( i = \{i_j\} \).
   It is a vector of the indices used to implement an operation \( i_j \), each of which is a label. Then length of \( i \) is equal to the object's rank. If the object is scalar, vector \( i \) has zero length. (This is naturally very similar to the definition of \( d \), the shape).

3. We introduce a shorthand for an expander (similar to the expander of page 213) for index vectors such that
   \[ g^x = E(i^x, m^x), \]
   It is then possible to transliterate every definition in F-code for shape into expressions for indices, in the following way:
   1. \( d \) becomes \( i \)
   2. \( e^x \) becomes \( g^x \)
   3. \( E(d^x, m^x) \) becomes \( E(i^x, m^x) \)
   4. \( p^x(d^x) \) becomes \( p^x(i^x) \)
   5. \( p^x(d^x) \) becomes \( p^x(i^x) \)
6. All arithmetic functions like max, +, etc. are re-named to conn

Secondly, the expression is flattened, such that the arity of an expression defines how many equivalences there are.

1. Example 1. If one considers the shape expression for dyadic, it begins \( d = \min(e^l, e^r) \), is transliterated to \( i = \text{conn}(g^l, g^r) \), and so equivalence between indices is \( i \equiv g^l \equiv g^r \). It is only the connectivity which is intended to be found from the original equation for shape. \( \text{conn} \) is an operation in which all of its operands are equivalent. So \( \text{conn}(g^l, g^r) \rightarrow g^l \equiv g^r \).

2. Example 2. If one considers the shape expression for reduce. Originally \( d = \text{pa}(d^a) \). It is transliterated to \( i = \text{pa}(i^a) \). The equivalence between indices is then \( i \equiv \text{pa}(i^a) \). This shows that there are a set of external indices for reduce, which are denoted by \( i \) and a set of internal indices for reduce, which are denoted by \( \text{pa}(i^a) \) which perform a reduce for an atom of the result.

3. Example 3. If one considers the shape expression for sect, it is defined in two parts:

\[
\begin{align*}
m_j &= \begin{cases} 
1, & \text{if } j \neq N \\
0, & \text{if } j = N 
\end{cases}, \quad 0 \leq j < r(d^4).
\end{align*}
\]

\( d = \text{P}(d^4, m) \)

The shape expression is transliterated to

\( i = \text{P}(i^4, m) \)

Then, due to the definition for \( m \), it is evident there is one index which is not externalized, which is the index of a sect which is to be fixed at a constant value. See definition for sect for details. The equivalence is then \( i \equiv \text{P}(i^4, m) \), with one internal index.

This is rather a laconic description of a more formal method for defining, or rather knowing the scopes of indices. An external index for a particular F-function is one which is visible to higher nodes in the F-tree. (Like those dimensions of reduce which do not take active part in the reduce, and all those indices except one which are part of \( i \) for sect). An external index for a particular F-function is one of those included in \( i \). An internal index is exactly the opposite: those indices which take active part in a reduce, denoted by \( \text{pa}(i^a) \), and that index which is fixed in a sect operation. Every internal index is the generation of a new index to perform some of the semantics of an F-code function.

It is now possible to define, implementationally, the scope of indices in an F-program. Within a lazy environment, where the following functions occur, new instances of indices are created to implement a certain facet of the lazy function. There are very few functions in F-code which generate, or operate on internal indices, those are: (the definitions are only true in the context of their definitions for shape)
reduce whose internal indices are $p^a(i^2)$ and whose external indices are $i \equiv p^a(i^2)$. The internal indices for this function create an index space across which the reduction takes place.

assign whose internal indices are $\text{conn}(i^1, p^r(i^r))$ and has no external indices. The internal indices for this function all create an index space across which the assignment takes place.

repl whose external indices are $i$, one index for each dimension of the result. Its internal indices are $i^s$ — one index for each dimension of $\text{EXPR.s}$. All internal indices are evaluated from external indices. (Some external indices are ignored). An example of repl's index manipulation is given on page 151.

comp whose external indices are the same as both as the indices of both the left and right, with perhaps one more external index, whose value differentiates left from right, above a certain threshold which is the extent of the left along dimension $N$. The extra external index is generated for the right operand, if $e^r_N \neq \infty$, otherwise there is no need to generate it at all. An example of comp's index generation and manipulation is given on page 150.

sect which has a single internal index $i^N$, which is fixed by the sect operation. This single internal index fixes one index of its operand at a particular value, thereby enforcing the lazy evaluation of only one layer of the operand. An example of sect's index manipulation is given on page 150.

slice which has a single internal index $i_M$, which implements the $\text{EXPR.i}$ of the slice operation. The value of the internal index is the dereference of $\text{EXPR.i}$ at position external index $i_N$, such that $i^s_N = \text{EXPR.i}[i_N]$. An example of slice's index manipulation is given on page 150.

transform has internal indices $i^s$, and external indices $\text{gen}(), 0 \leq i < n$; at run-time, values of internal indices are derived from external indices; there is no external index which has any bearing on $\text{EXPR.s}$. An example of transform's index manipulation is given on page 151.

pack has internal indices $i^d$, and external indices $i$ generated to represent $d$; at run-time, values of internal indices are derived from external indices; there is no external index which has any bearing on $\text{EXPR.s}$. An example of pack's index manipulation is given on page 152.

gather has internal indices $p^d(d^4)$ along which to the gather, and external indices $A(p^d(d^4), \text{gen}())$, where $\text{gen}()$ generates a new index, and $A(v, x)$ is vector $v$ with the component $x$ appended
• pol has internal indices \( i^e \) and external indices \( i \) generated to represent \( d \); at run-time, in order to evaluate an atom of \( c \), \( \text{EXPR.} \ c \) has to be scanned in totality; no index on which \( c \) depends has any effect on an external index.

These operations between external and internal indices do one of two things:

1. internal indices may depend on the value of external indices, or arguments of the function — repl, comp, sect, slice, transform, pack

2. internal indices do not depend on external indices or arguments of the function, but rather are used to implement internal loops — reduce, assign, gather, pol

When internal indices are created, they have a certain meaning in implementing the F-function they are the internal indices of. For reduce, assign, gather and pol they are used to create or augment the lazy invocation of their associated function with an index space across further dimensions. For reduce, they implement the index space which performs an elemental reduce. Thereby the reduce itself is invoked lazily producing a scalar result elementally across the dimensions of the reduce which do not perform reduction. For example: (reduce 110 ...) has two active dimensions which reduce across a matrix, and one inactive dimension provided by an external index. The external index is then used for further laziness: only elements of it which affect the result need to be evaluated. For example (slice (reduce 110 ...) ...) may select only a number of the results of reduce, along the external index of the reduce. Examples of sect, slice, comp, transform, repl, and pack are given later in section 5.3.5 (—this section may be read immediately.)

There is a final case for the introduction of internal indices which is that for hold. The internal indices are \( i^e \), where \( \text{EXPR.} \ i \) is the ‘held’ object. It needs indices of its own (internal indices) since \( \text{EXPR.} \ i \) is evaluated independently of \( \text{EXPR.} \ a \); and is evaluated in advance, as a ‘function’ or ‘common-subexpression’ in its own right. This is not really an operation on indices.

4.4.2 Scopes of indices

The previous section introduced some notation for indices. Its intention is to show the scopes of internal and external variables. The scope of an internal variable is the daughter nodes of the function which introduces it (as such, only the functions above can introduce internal variables). The scope of the index can be limited far better than that in practice — to only those functions which depend on it, in a path from this node to a piece of data, or reference to a piece of data.

External indices with respect to one function may be internal indices with respect to a function higher up the tree — consider the internal loop indices of a reduce are external indices to the object it is reducing. Finally, at the very top of an F-tree are a set of external indices which are not internalized at all, and these are the primary indices of the tree, which evaluate the full result of the entire F-expression, or subroutine.
4.4. INDICES

In this figure, \(i_2\) and \(i_3\) are internal indices for the reduction, \(i_1\) is an external index for the entire function.

Figure 4.8: Example of index labelling

4.4.3 Shape trees and indices

Expressions from indices can be transliterated quite simply from shape expressions. In the previous chapter it is shown how shape trees are created along side the original F-tree. There are, at every node of the primary F-tree, an array of integer scalar F-trees each of which evaluates to the extent of a particular dimension of the primary F-tree.

If one sees the connectivity of these trees, if nodes in it are not folded to constants where they can be, one sees that the shape trees give a route for distributing (labelling/numbering) indices. This is best demonstrated by example. Generation and propagation of indices, regardless of its complex definition, is simple; this is just a facet of the generation of shape expressions (from the previous chapter).

In an example of index labelling:

\[
(\{i_1\} \text{dyadic mul}) \\
(\{i_1\} \text{reduce } 110) \\
(\{i_2,i_3,i_1\} \text{dyadic add}) \\
(\{i_2,i_3,i_1\} \text{var value A}) \\
(\{i_2,i_3,i_1\} \text{call}) \\
(\{i_1\} \text{var value C})
\]

where

\(X_6,X_7,X_8,X_9,X_{10},X_{11},X_{12}\) are integer constants or variables

\(X_6,X_7,X_8,X_9,X_{10},X_{11},X_{12}\) are integer constants or variables
\[ X_3 = (\text{dyadic min} \ X_6 \ X_9) \]
\[ X_4 = (\text{dyadic min} \ X_7 \ X_{10}) \]
\[ X_5 = (\text{dyadic min} \ X_8 \ X_{11}) \]
\[ X_2 = (\text{monadic ref} \ X_5) \quad \text{% since the mask is 110} \]
\[ X_1 = (\text{monadic ref} \ X_1) \]

becomes (since these inferences produce a tree for index distribution):

\[
\begin{align*}
([i1] \ \text{dyadic mul} \\
([i1] \ \text{reduce} \ 110 \\
\quad ([i2,i3,i1] \ \text{dyadic add} \\
\quad \quad ([i2,i3,i1] \ \text{var value} \ A) \\
\quad \quad ([i2,i3,i1] \ \text{call} \ (\text{var value} \ B) \ \text{integer 3 value})
\end{align*}
\]

This is shown in figure 4.8.

Here, \( i_2, i_3 \) are internalized indices, which are used to atomically evaluate a reduce, and \( i_1 \) is an externalized index which is altogether independent of the process. The algorithm to number indices is trivial: just to allocate a new number for every dimension of the operation and then to traverse the relevant extent tree, attributing each node of the extent tree with this index number. This tree-traversal terminates either on reaching a leaf, or reaching a node which is already numbered. A node may already be numbered not because the extent tree can ever be cyclic — extent trees are always acyclic — but because of other lazy environments which are numbered in advance.

In terms of index notation, for the reduce, the internal indices are \( p^a(i^a) \), which is \([i_2, i_3]\), and the external index is \( \bar{p}^a(i^a) \), which is \([i_1]\). \( i_2 \) and \( i_3 \) are new indices which need to be created to execute the reduce.

Figure 4.9 shows some more examples of relationships between indices. The reader might recognize this figure from figure 3.3 or might like to refer back to it.

In the left half of this figure, \( i_2 \) is an index applicable to the entire function, and \( i_1 \) is an index applicable to the assignment (which assigns elements of \( A \) indexed by \( i_1 \) with the scalar value 0).

The right half of the figure is more interesting. \( i_3 \) is the only external index of the function. The indices are not independent, and a calculation is made appropriately scheduled in the following way:

\[ i_2 = (\text{ramp} \ (\text{const} \ 0) \ (\text{const} \ 8) \ (\text{const} \ 2)) \ [i_3] \]
\[ i_1 = 0 \]

Applications of this kind of index manipulation will be given in the chapter on code generation (see section 5.3.5.)
4.4. INDICES

4.4.4 Folded shape trees

The routine to fold constant subexpressions, given in section 3.5, describes the simple algorithm which evaluates constant (scalar) expressions and re-writes the tree into a constant. Previously, it was shown that in order to distribute indices, one makes use of the extent trees, by merely following their connectivity. If constant sub-expressions were simply folded, this tree-effect would be lost. It is necessary, even when trees are folded to constants to retain a pointer to the daughters (which were folded), so that index distribution can follow the same path and the extent expressions can be folded.

By simple example:

\[
\begin{align*}
\text{([8] dyadic max} \\
\quad (\text{[9] const "Pessimism"}) \\
\quad (\text{[8] const "Optimism"})
\end{align*}
\]

The extent tree for the max was evaluated from \(\text{dyadic min 9 8}\), and was folded to just the value 8 (thereby losing the reference to its two operands). If one attempted to distribute indices, the following would happen:

\[
\begin{align*}
\text{([11] dyadic max} \\
\quad (\text{[?] const "Pessimism"}) \\
\quad (\text{[?] const "Optimism"})
\end{align*}
\]
Chapter 4. Sequential Aspects of F-Code

This is a purely implementational aspect, but an important one to realize to make the implementation easy. If the pointers are retained, the index can be propagated, as it should be, to the leaves:

```plaintext
([if] dyadic max
  ([if] const "Pessimism")
  ([if] const "Optimism")
)
```

A constant in the extent trees must be represented with a node like `{ (const 8) L R }`, where L and R were the operands of what was folded to a constant; or they may, either or both, be \(\bot\) if it/they was/were not folded from a binary operation, or not folded at all.

4.5 Sequential ordering of an F-code tree

The sequential ordering given here has no output, but is employed by the code generator to produce a sequential ordering between lazy environments (and shows the laxity of ordering with lazy environments). This sequentialization is as applicable to distributed machines as to a sequential machine, because it does not delve into low-level (fine-grain) scheduling issues. For example: The F-code function \(\text{seq} \ A \ B \ldots\), the sequentialization of \(\text{A then B then} \ldots\) must of course be implemented \(A ; B ; \ldots\) on all machines, where `;` is the symbol for sequentialization, as usual. The functional subset of F-code — that part of F-code which forms lazy environments — give a compiler a large amount of freedom for optimization as will be seen in chapter 5.

F-code programs can be arbitrarily complex: F-code is far more general than a high level language and there are cases which simply are not expressible in high level languages which can be expressed in F-code. One has taken the abstract approach, discarding all ideas of high level languages, and merely tried to produce a concrete implementation of the data-parallel semantics, and dynamic scoping mechanism, of F-code, in the knowledge that this subsumes the modes of operations of high level languages.

It is necessary to define, quite formally, in order to make the conceptualization of the process easier, the sequential ordering of an F-code tree. There are functions in F-code which can be executed lazily, which are grouped together in a lazy environment; there are those which have side-effects (`put, template, create, coerce, mark, and call`); and those whose purpose is solely to impose (or provide laxity of) sequential ordering (`seq, par, etc.`)

This section deals with these three kinds of functions in turn, and goes on to give a full example of the coarse-level sequentialization of an F-code program.

Lazy functions: Lazy functions are arithmetic expressions, geometric expressions — all those which do not include side-effects (in the F-code sense). Lazy evaluation in this sense is the ability to evaluate only necessary parts of a computation spatially (across the iteration space). This has already been described.
Lazy functions are conceptually grouped together into lazy environments, see section 4.3.1. A leaf of the lazy environment is either the node above an eager F-function, or a leaf of the F-tree if there is no intervening eager function.

The lazy environment may contain commas (side-effects), and the execution of a lazy environment is thus the following chain of events:

1. Side effects may occur before the execution of the lazy environment. All comma right functions in the tree trigger side-effects which must be evaluated before the execution of the environment itself — and thereby may affect operands of the lazy environment.

2. The operands of the lazy environment must then be computed — in any order. They exhibit functional concurrency. (Operands may be eager functions and other lazy environments).

3. The body of the environment is executed eagerly: laziness is provided by the fact that its iteration spaces are the most selective possible. The environment may contain internal indices and so forth described in the previous section.

4. The data allocated for operands of the lazy environment must then be discarded.

5. Side effects may occur after the lazy environment. All comma left functions in the tree trigger side-effects which are evaluated after the execution of the environment.

Side-effects in this case are assignments. The ordering of commas can make the program non-deterministic. While commas towards the top of the environment are executed before those towards the bottom (side-effects are evaluated depth-wise, top-first), functional parallelism may induce non-determinism in side-effecting environments. The compiler will make no attempt to detect or alleviate this because if the F-code program specifies something non-deterministic, the compiler merely implements it as such.

The body of the environment is executed eagerly. But as yet, there is no need to deal with fine-grain scheduling issues. The functional parallelism of the body is passed through to the code generator, which schedules it for a congruent implementation on a RISC processor — by using the functional parallelism for pipelining, etc. The sequential ordering specified above for lazy environments is at a coarser level, which says such-and-such must occur before such-and-such to conform to the semantics of F-code; and there is no freedom at this level at all. The five steps above must be sequentialized.

**Side-effecting functions**: Beside comma which is used to integrate side-effects into a lazy environment are the eager functions such as create which impose further sequentialization in a program. The full list of these functions is put, hold, template, create, coerce, mark, and call. hold is very similar in reality to comma left, and in a future version of the compiler it may be implemented as such: it holds its argument in a variable name for
the scope of a particular function. At the moment, however, it is implemented as a purely
eager function, in the same way as other eager functions, described below.

Eager functions require their arguments to be computed in advance, and go on to trigger
the evaluation of their main operand (or in the case of call, call a function). Fundamentally, this sort of side-effecting function goes against the grain of lazy evaluation. It
cannot be executed inside a lazy environment; these eager functions generally bound lazy
environments within a program.

The order of execution of a side-effecting (eager) function is:

1. Evaluate the parameters of the F-function. (For example, parameters of create
define the shape of the object to be created). And perform any side-effect. (For a
create, this creates the object.)

2. Evaluate the main operand of the F-function. The operand may be eager functions
and lazy environments — or in the case of call it is the body of a function to be
called (via a pointer.)

3. Do any side-effects which occur after the evaluation of the main operand. (For ex­
ample, for create, the object created in step 1 is now destroyed).

The full ordering of a data-parallel F-code tree has now been shown, if one composes
the order of evaluation of side-effecting functions and lazy environments in relation to a
particular program.

Sequential: The final sort of F-function which affects ordering are the most trivial to im­
plement. These are the seq and par functions of F-code. They only return scalar re­
results, and only invoke functions which return scalar results. They are easily compiled:
(seq A B ...) is $A ; B ; ...$ and (par A B ...) is compiled, depending on the archi­
tecture, as a parallel or sequential invocation of $A, B, ...$

In the sequential implementation, par may be [and in the current compiler is] implemented
almost identically to seq: it sequentializes its operands.

4.5.1 Example of coarse-level sequentialization

This section intends to give an example of the sequential ordering of an F-code tree, including
finding lazy sections within a tree and subsequently being able to order it.

Let us consider the following program, and its F-code equivalent. The function $- (B(A=0) \cdot 2)$,
where $A$ is a matrix of shape 100 x 100 may be represented by the F-code function below. $A=0$
is used to zero the entire matrix, passing the matrix as parameter to $B$. The scope of $B$ in the
program is not shown; the program assumes the variable $B$ contains a valid pointer to a function.
4.5. SEQUENTIAL ORDERING OF AN F-CODE TREE

a: (create A
b: (dyadic neg
c: (comma right
d: (assign
e: (var name A)
f: 00 (const 0)
 )
g: (dyadic mul
h: (call
i: (var value B)
j: integer 2 value
 )
k: 00 (const 2)
 )
 )
l: integer
m: (const 10)
n: (const 10)
 )

The program is annotated with letters to aid discussion on the sequentialization of this program. Side-effects of eager functions are denoted by this letter with superscript 6 for before side-effect, and  for after side-effect. Thus a is the before side-effect of create, which is the creation of a 10 x 10 matrix, and a is the after side-effect of create, which is the destruction of that matrix.

First of all, let us find the lazy environments in this program, according to the method described at the beginning of this chapter. This results in the conceptual partitioning shown in figure 4.10.

The ordering of this tree is described below in terms of the annotation. In the expressions below, square brackets are used to identify execution of the bodies of lazy environments, ';' identifies sequentialization within a lazy environment, '|' identifies possible parallel execution within a lazy environment, and '!' identifies global (control) sequentialization. Bold letters (b, d, etc.) mean the execution of an entire F-tree, normal letters (b, g, etc.) mean the execution of a particular F-function. There is also a need to express the destruction of anonymous data (operands to lazy environments), which are bold letters with the superscript * (such as b*, d*, etc.)

a is an eager function, create, which has the ordering (m!n!a!b!la). That is, it evaluates the shape of the matrix; a creates the matrix; it evaluates the main operand b; and then a destroys the matrix.

b is a lazy environment, which is sequentialized in five stages: evaluate the discarded operands of comma rights, evaluate operands of the lazy environment, evaluate the body of the environment, destroy data allocated for the operands of the lazy environment, and finally evaluate the discarded operands of comma lefts. This results in the ordering
This lazy environment is a side-effect of the COMMA RIGHT.

Figure 4.10: Finding the lazy environments

\[ d \text{!} h ! [(k; g; b)! h^*]. \] The operand of lazy environment \( b \) is \( h \). The destruction of the data of this operand is \( h^* \). The side-effects of the lazy environment are contained in \( d \). The body of the lazy environment \( [k; g; b] \) also includes a reference to the result of the call, a reference which may take place in parallel to \( k \). If this reference is called \( h' \), the proper ordering is \( d \text{!} h ! [(k|h'); g; b|h^*]. \) This shows that \( h \) is evaluated before the lazy environment, it is referenced in the lazy environment, and it is destroyed afterwards. This shows an example of the scope of an anonymous variable in the evaluation of the F-tree.

\[ d \text{ is another lazy environment whose execution is merely } [(e|f); d]. \]

This means that the entire sequentialization is automatically:

\[ (m|n)! a^*! [(e|f); d]! h! [(k|h'); g; b]| h^*! a^* \]

The evaluation of F-code is thus the interlaced operation of global control operations, and lazy environments (which are data-parallel).

This section has shown the coarse-level sequentialization of an F-program. Now it is useful to describe when shape expressions may be evaluated, and hence how lazy evaluation is carried out inside lazy environments.

If shape expressions are scalar constants, it does not matter when they are evaluated, but when shape expressions are include variables, because the shape of a lazy environment's result
is dynamic, the extent variables must, of course, be written before they are read.

Inside any lazy environment, one ought to notice that rewrites for dynamic shape (section 3.7.3, in the previous chapter) rewrite the lazy environment by including a comma right which is used to introduce an assignment to an extent variable. This section has already shown the fact that lazy environments are ordered by: (1) evaluating comma rights; (2) evaluating the operands of the environment; (3) evaluating the environment's body; (4) destruction of operands; (5) evaluating comma lefts. Therefore, the extent variables are automatically written to before the body of the environment is executed. Therefore, the extent expressions may be evaluated (to find the shape of a lazy environment) just before the body of the environment is executed, which is perfect. In the expressions above, [..] denotes the lazy evaluation of a piece of the F-tree. For every one of these, the extents may be evaluated beforehand, which one could denote by a superscript, such that \([b]^6\) means the body of lazy environment \(b\) is evaluated after \(b\) which are the shape expressions.

The full sequentialization is thus, including the evaluation of the shapes of environments:

\[(m|n)!a^4![(e|f); d]^4!hl[(k|h^6); g]; b]^4!h^6!a^4\]

The elemental operations within lazy environments are independent: they may occur in parallel. This may induce non-determinism in an implementation where the lazy environment is an assignment, and parts of the target are duplicated. There is no attempt to alleviate this. The program given to the compiler in this case is merely garbage. The ordering shown inside a lazy environment, such as \((k|h^6); g; b\) is the elemental case. Thus a particular element of \(k_i\) must take place before \(g_i\). The shape of the index space \(i\) is determined by the shape expressions of the lazy environment.

This section has shown the coarse-level scheduling of F-code, where there is no laxity in the implementation: these orderings are necessary. Within lazy environments, since elemental operations are independent, there is full laxity of ordering across the index space.

### 4.5.2 Comparison between lazy environments and I-structures

A lazy environment is similar to some kinds of I-structure [ANP89] from the language Id. The I-structure:

```plaintext
{ A = Lmatrix ((1,m), (1,n));
  { For i ← 1 To m Do
    { For j ← 1 To n Do
      A[i,j] = i+j } } } in A
```

is equivalent to the F-code lazy environment:
This lazy environment is dynamically shaped (it depends on m and n). The result of rewrites that occur on this program because of this dynamic shaping is:

\[
\begin{align*}
&\text{( dyadic add } \\
&10 ( \text{ ramp } \\
&\quad ( \text{ const 1 } ) ( \text{ const m } ) ( \text{ const 1 } ) \\
&01 ( \text{ ramp } \\
&\quad ( \text{ const 1 } ) ( \text{ const n } ) ( \text{ const 1 } ) \\
&\text{) )}
\end{align*}
\]

This is sequentialized to do the following:

1. Evaluate all the discarded operands of comma right. That is \(_x1 = m\) and \(_x2 = n\).

2. If the environment had operands (which it hasn't) these would be evaluated. (Which could modify the values of \(n\) and \(m\)).

3. Next the body of the environment is executed, whose shape is \(<_x1,_x2>\), writing to a piece of data that size.

4. If the environment had operands, their data is now destroyed.

5. If the environment had comma lefts, their discarded operands would then be evaluated.

This results in the following sequentialization, after the indices have been labelled \(i1\) and \(i2\):

\[
\begin{align*}
&_x1 = m; \_x2 = n; \\
\text{result } &\leftarrow \text{alloc_mem}(_x1,_x2,\text{integer}); \\
\{ \text{ For } i1 \leftarrow 0 \text{ To } (-x1-1) \text{ Do } \\
\quad \{ \text{ For } i2 \leftarrow 0 \text{ To } (-x2-1) \text{ Do } \\
\quad\quad \text{result}[i1,i2] = (1+(i1*1))+(1+(i2*1)) \} \} \\
\text{return result; } \}
\end{align*}
\]
4.5. SEQUENTIAL ORDERING OF AN F-CODE TREE

This memory write for this loop can be optimized on a sequential machine to: (as described at
the beginning of this chapter)

\[
\begin{align*}
\text{pointer} &= \text{result} = \text{alloc\_mem}(x_1, x_2, \text{integer}); \\
\{ &\text{ For } i_1 \leftarrow 0 \text{ To } (-x_1-1) \text{ Do} \\
&\{ \text{ For } i_2 \leftarrow 0 \text{ To } (-x_2-1) \text{ Do} \\
&\quad \ast\text{pointer++} = (1+(i_1*1))+(1+(i_2*1)) \} \} \}
\end{align*}
\]

— borrowing the syntax of C for post-incrementing writes to memory (via a pointer).

The point of I-structures is that as soon as the result is allocated, it can be returned; elements
of the result maintain the flag \textit{empty} or \textit{full} so that they cannot be read before they are written.
This permits a certain amount of pipelining between data-structures: one may read elements of
another while its elements are being computed.

What bounds lazy environments in F-code are eager functions, like create. The current
implementation assumes that the entirety of the result is computed before it is returned, thus
precluding pipelining between almost adjacent lazy environments, sequestered by the execution
of the eager function. This does not affect the efficiency of a scalar implementation of F-code.

4.5.3 Pipelining between almost adjacent lazy environments

This is projected work for a distributed implementation of F-code. While it is complicated,
pipelining seems to be possible between almost adjacent lazy environments in an F-code program.

If they were adjacent (touching) the two lazy environments can always be combined into one
lazy environment. Thus they are always almost adjacent, sequestered by eager, mainly scoping-

system, F-functions.

If these eager functions did not exist, and an F-tree consisted solely of lazy environments
connected together, it is quite easy to see that elemental results of one lazy environment can
be evaluated as soon as its elemental operands are available. This is data driven data-flow in a
lazy parallel setting.

It is not as straightforward as this in an F-code program where eager functions do exist. The
eager functions provide scopes of variables for the lazy environments which they bound. While
elemental results are still being computed of a particular environments, the scopes must remain
valid. This means that there can be multiple instances of a variable active at the same time:
the current implementation of the dictionary as a stack is not valid. Like multiple invocations of
functions, where parameters and variables whose scope is inside the function must be duplicated,
the implementation of pipelining may mean that variables of the same name used in adjacent
lazy environments must be duplicated.

This might pose no problem: the implementation of the parser must flatten the variable
name space: while variables share the same lexical string, they are allocated different entries in
the symbol table, and hence in the dictionary. Thus there will not in effect be two variables of
the same name in a program. This, of course, means a larger dictionary will be used. The effect of this imposes on the dictionary can therefore be ignored, since it can be removed with textual analysis.

Let us associate a lazy environment with all the eager functions above it, up to the next lazy environment, and consider this as a unit — and call it an F-segment. Any F-program is composed of a number of F-segments. The F-segment has a number of events associated with it: the events depend on whether or not the F-segment contains any comma lefts.

If the called segment does not contain comma left:

1. Trigger any number of before side-effects of all eager functions at the top of the F-segment.

2. Trigger any number of before side-effects because of comma rights inside the F-segment. This means that the shape of the lazy environment component of the segment can be evaluated.

3. Evaluate the shape of the lazy environment's result. Allocate a result piece of memory (with all elements tagged as empty). A pointer to this piece of memory can be returned immediately.

4. Trigger the operands (other F-segments) of the lazy environment, in any order. For a pipelined implementation, they return a pointer to a piece of memory which will contain the results. Each element of the memory has a full/empty flag. Stages 3 and 4 can be done in parallel.

5. Pipelined, lazy execution of the body of the lazy environment begins immediately — before all or any of the data has been returned from the operands.

6. When the execution has completed, all the operands can be terminated, and this results in the following:

7. When the F-segment gets a signal to terminate, it destroys the piece of memory where it stored its result. It also triggers any number of after side-effects of all eager functions at the top of the segment.

If the called segment does contain comma left, it cannot return its result in a pipelined manner because of the side-effects it contains which must be executed after the segment has been evaluated in full. This order is as follows:

1. Trigger any number of before side-effects of all eager functions at the top of the F-segment.

2. Trigger any number of before side-effects because of comma rights inside the F-segment. This means that the shape of the lazy environment component of the segment can be evaluated.
4.6 Summary and conclusions

This chapter has described some of the aspects of sequentializing F-code, not necessarily to a sequential machine, but rather it was a chapter on how to adhere to the necessary dependencies in F-code.

1. Firstly, the process of loop filtering is used to infer the positions of data-parallel loops in the F-code tree; and the positions of lazy environments. Secondly, it described data-parallel loops, and how they may be optimized on a scalar machine. Positions of data-parallel loops indicate the tops of lazy sections of F-code.

2. A method of denoting indices was used which given an F-code specification can denote which index applies to which node, using a similar mathematical equation to the shape equation, by a simple transliteration. The notation is thus a method of describing the internal and external indices used to implement any particular F-code function.

3. How F-code programs are sequentialized has also been shown, with an example. This is the coarse-level sequentialization of F-code programs. It is shown that the shapes of lazy environments may be pre-evaluated before the execution of the lazy environment. This is very important for efficiency.
This has shown the compilation of F-code programs to be quite trivial, using the shape expression trees which have already been constructed by shape inference.

The tree is now ready for code generation. The next chapter describes the run-time environment which is to be assumed for a real implementation. In order to make the implementation architecture-restricted, but still general across a range of processors (in this case all RISC processors) the code-generator which has been implemented (called the targetter in Portable Software Platform terms) creates a data-flow graph, still retaining the explicit notion of loops.

It is possible to stop the thesis here: the compiler at this point can produce scalar C code. One can argue therefore that with this amount of work (and a quite simple C code generator), it is possible to compile F-code to any machine, since C is portable. The introduction of this thesis, however, said that C is not architecture-neutral. Any C program as it stands cannot be implemented with optimal efficiency on all machines (distributed machines), because it lacks information. The route taken instead, uses a further lower-level intermediate platform called T-code. To reiterate, T-code answers the question: why discard anything (a tractable representation of parallelism, including known dependencies, etc.), and consequently attempt to re-infer everything that was discarded. One would require a very sophisticated C compiler, which can infer parallelism in its original form, if C were taken to be the intermediate level.

### 4.7 Applying this chapter to parallel machines

The major difference between a sequential implementation and a parallel one is the fact that a 'parallel' computer does not need loops, for loops are dualed in the parallel domain by a number of mechanisms like communication actions (for such functions as reduce) and data-distribution. Loops in this sense are the loops required to implement lazy environments.

A parallel implementation of F-code may be a reticulated set of sequential compilations bonded together by inter/intra-processor communication. As such, the compiler for F-code to a purely sequential machine should always be the first stage in compiling it to a parallel machine. It is quite standard in this respect. Recapping, the F-code tree to this point has inferred the following: (1) type, rank and sort attributes for every node of the tree. (2) The shape of every function in the tree (which may include variables in the dynamic case.) (3) The positions of loops, hence lazy environments which an F-code program can conceptually be chopped into. These environments may be executed in a pipelined way.

The positions of loops in the parallel domain may be taken to mean: bulk synchronizations at the top of lazy environments. Elemental computations in a lazy environment are independent. All of the atomic operations of the 'loop' may be 'forked' and executed concurrently and only 'rendezvoused' at the end. The coarse-level sequentialization discussed for F-code on a scalar machine is equally applicable to that on a distributed machine. This demonstrates quite well how applicable F-code is to both scalar and parallel machines.
Chapter 5

Code generation

Figure 5.1: Stages of the code generator

- 5.1 — Introduction ................................................................. 128
- 5.2 — T-code: A machine-independent graph for targeting .................. 129
- 5.3 — Transforming F-code into a T-code graph ................................ 140
5.1 Introduction

F-code, as a PSP, has high-level semantics, such that the translation procedure to it from high-level programming languages is straightforward. Compilers to F-code do not need to include optimization because this is part of the F-code compiler, since this is what F-code is very good at specifying and providing; compilers for scalar languages need prior vectorization, however. In fact, F-code could be treated as a black box: a system of data-parallel optimizations which may be increased over time, rather like the additions made to Spice, the circuit simulation package. One need not know any of the frippery which occurs inside, merely the interfaces.

F-code was defined in terms of data-parallel algebra, which also becomes a useful manner of specifying transformations. Up to this point, optimizations such as pipelining, or those allied to inter-processor communication, or load-balancing have not been considered, but rather this is the representation of a computation which requires fewest instructions, or processors or whatever one intellectually decides to be a unit of computation without basis in physical reality at all.

Figure 5.2: Subordination of a targetter to a machine-independent front-end

The figure above suggests that a compiler should be partitioned into one or more parts. The reason F-code exists is that there are very many high-level languages, which although they appear to be very different indeed, have only syntactic difference. What underlies most imperative programming languages is data-concurrency and it is this which is the basic notion: operations operate atomically over a data concurrent object. This is to say nothing about the machine, or how it operates, or how the language is implemented.

The basic premise is that after syntactic distortions are removed from high-level languages, one always results in something very similar: a data-parallel PSP. Therefore there can be no reason to ever rewrite the front-end of a compiler for a PSP, since it is apposite for any high-level
5.2. A MACHINE-INDEPENDENT GRAPH FOR TARGETTING

language. Facets of the front-end of the compiler occupied the previous two chapters — they are all machine-independent aspects of compiling F-code. What remains is to target the already much-rewritten parse tree to a real machine, and this is entirely, of course, machine-dependent. It seems logical to partition the compiler at this point into a front-end and a targetter.

This chapter is limited to a certain class of implementation. The partition is valid for any implementation, however the second layer (the targetter) intermediate languages may be fundamentally different.

1. The target-machine is probably a load-store oriented (RISC) machine

2. The target-machine is probably NOT a vector machine

'Probably' because where machines are not RISC, are not load-store oriented, or are vector machines, these machines can still be catered for, but with difficulty because this is perhaps not the most apt approach. This chapter is limited to these types of machine, however it is still appropriate for a distributed compiler since as previously repeated, this sequential compiler is ancillary to the distributed one.

If the compiler is to be partitioned it needs an interface language — this, since the intention is now to compile to a scalar machine, can be a purely scalar language, with loops, suitable for fine-grain scheduling and optimization. It can be somewhat like a regular assembly language. One asks the question: 'is F-code suitable to be this language?' In other words: 'is F-code equally applicable both for high-level and low-level optimizations?' The answer is probably not: the second-layer (targetter) intermediate language needs to be near assembly language which F-code is not. Unlike assembly languages however, it must be architecture-neutral.

The targetter's intermediate language — call it T-code — must include all of the mathematical and geometric operators of F-code but they need only apply to scalars. Geometric operators utilize scalar indices. In order to represent a conceptually data-parallel program, the language now includes loops explicitly, which provide incremental values for indices. It is quite like vectorized FORTRAN. The graph must be able to represent the scalar equivalent of an F-code computation with an identical functionality. F-code is a tree, close to the syntax tree of the original source language, representative of operational semantics and data-parallelism, and the tractable representation of scope. Real compiler fine-grain tasks such as register allocation and life-time analysis, scheduling and pipelining, etc. are more naturally done in terms of graphs. Since F-code does not include 'goto' it renders it unusable in assembly language terms. The intermediate language should be a graph, whose arcs are purely scalar (which may be scalars, or pointers to blocks of memory).

5.2 A machine-independent graph for targetting

The operations in the machine-independent graph are all scalar, however data-concurrency is not discarded but remains represented by [nested] loops. In fact, from the representation of
loops it is still possible to produce a distributed implementation, if the implementor chooses how to distribute a particular ‘loop’, executing it in parallel and not sequentially; this is not the aim, since the graph is intended for a single processor.

The basic unit of the graph is a triadic node which is labelled with a particular semantic role like add, loop, seq, par, etc. The graphs are capable of describing data dependencies and also necessary control dependencies. The data dependencies are used to evaluate lazily, and the control dependencies are used to place necessary sequentializations. Data-antidependencies are not represented explicitly, except by means of control dependencies. The reason that nodes are triadic is to give a node one destination and two sources hence to represent three-address-mode types of instructions. The approach of using graphs for targeting is most suited to register/stack-based RISC machines since the arcs in the graph can generally be implemented using registers, if the machine has enough registers.

There is no definite sequential ordering of the graph, unless the role of a node enforces sequential ordering explicitly like seq does — this is used to represent all necessary sequentializations which were described in the previous chapter. Scheduling the graph involves transforming the graph into a graph with a spine. The spine is the sequential ordering of instructions in the graph, which will take into account efficient uses of execution pipelines, perhaps memory pipelines and arithmetic pipelines — in fact all of the instruction-level hardware concerns.

5.2.1 F-code types, machine types

The graph is effectively a program to execute on an abstract machine. The abstract machine is given the following machine types:

**Byte**: A byte is of a sufficient range of values to hold character constants (characters are unsigned short integers)

**Word**: A word, generally the word-length of the machine, is of a sufficient range of values to hold positive and negative integers

**Float**: A float may hold a real number (one does not assume any standard format for the representation of floating point numbers)

**Complex**: A complex may hold a complex number, and could be a pair of floats

The machine types correspond to the F-code types in the following way:

<table>
<thead>
<tr>
<th>Logical</th>
<th>→</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>→</td>
<td>Byte</td>
</tr>
<tr>
<td>Integer</td>
<td>→</td>
<td>Word</td>
</tr>
<tr>
<td>Real</td>
<td>→</td>
<td>Float</td>
</tr>
<tr>
<td>Complex</td>
<td>→</td>
<td>Complex</td>
</tr>
</tbody>
</table>
The type ‘logical’ has been discarded, at least for the moment, since most machine languages do not cope with bit-wise operations very well, and normally assembly languages do not include bit operations at all. Logical values are thus assumed to be held in bytes (the values for true and false are yet to be specified).

Types are not associated with any numerical system. Register telescoping (sign-extension) is not assumed to be a function of the hardware, thus where sign-extension is required, it is explicitly a node in the T-code graph. All permutations of type coercions need to be given a name and may be used as nodes in a T-code graph; they have roles like byt (truncate word to produce a byte), wrd (sign extend byte to produce word), fconv (integer to floating point conversion).

5.2.2 Abstract machine instructions

Each instruction is a node which has up to three arcs, of which normally one is a destination and two are sources. There are some variants on this: three sources; one destination and one source; one source; etc. Nodes are triadic (never more) because this is the minimum number of arcs a reasonable choice of nodes can have. All nodes are at most triadic including if (if a then b else c) which is purely a control node.

The instructions are formally defined in appendix F which the reader may need to refer to occasionally during this chapter.

The abstract machine is load/store oriented (fundamentally RISC) meaning most operations have register operands, and return results registers, with load/store operations being used to reference and write to memory as rarely as possible.

No assumption is made by the abstract machine of register telescoping after loads. Type conversions are placed explicitly. When a correspondence is made between abstract instructions and real machine instructions, if type conversion is made automatically by the hardware, the abstract instruction is ignored (and deleted). Placing conversions explicitly means that if the hardware does support them automatically, they may be omitted, and if the hardware does not, they must be emulated.

All operations work with scalar data. Operations correspond with the functionality of F-code in a scalar way, including operations to implement all intrinsic mathematical functions of F-code: sine, cosine, etc. The abstract machine uses a data-flow graph, except it assumes that some memory system and heap resources do exist (these are discussed later.)

Abstract control instructions include IF THEN ELSE and LOOP to emulate data-parallelism, in a similar way to DO loops.

5.2.3 Control dependency arcs

Control dependencies denote the strict flow of control of a program, and in T-code. No attempt has been made to turn control dependencies into data dependencies. A T-code graph integrates
control and data dependencies into a single graph. Since F-code is a PSP for imperative languages, and hence T-code contains explicit load and store operations to memory, it would be nonsensical to turn control dependencies into data dependencies.

Control dependencies are singly directed, because their use in scheduling is unambiguous: That is they say such-and-such occurs before such-and-such. The control nodes are used to implement the necessary sequentializations discussed in the previous chapter.

Figure 5.3 shows a number of examples of control dependency graphs. A tree providing control for four sequentialized subtrees (The four are to be executed in the order 1 < 2 < 3 < 4); a forever loop, which once started repeats for ever (it executes FOREVER until the program is terminated abruptly); finally, an example of the T-code node if which implements a choice (IF condition DO something ELSE DO something else). The if does exit, if its test happens to be false, or otherwise does seq. The test of the if is a data-dependency graph, and produces a logical result. if is purely a control node. It is triadic, and corresponds roughly to the assembly language version of if-then-else which has one data arc (the test), and two control arcs. It will be shown in figure 5.6 how a data version of if can be constructed from if itself, which computes a test and returns the result of one expression or another depending on the test.

5.2.4 Data dependencies

These instructions are connected by data-dependency and in some cases control dependency arcs. Data dependency arcs may be regarded to be communication paths, via channels or registers.
The full T-code graph is a mixture of data and control dependencies. This will be discussed in more detail later. Data-dependencies are also a form of control dependency, in that the operands of any node must be evaluated before the node if the graph is to be compiled to execute eagerly.

Figure 5.4 below gives a number of examples of data-flow graphs, where the control dependencies are implicit. They are not unlike standard parse trees, for scalar machines. Every arc is capable only of containing a scalar, and so these are not unlike standard data-flow graphs. In figure 5.4, dotted arrows depict the general direction of data-flow.

Figure 5.4: Example data-dependency graphs

If one considers register life-times for a single scalar processor, they exactly match one arc, however, when the graph is given a spine (scheduled), these arcs may envelope a number of other nodes. If the nodes correspond to machine instructions, the scope of a register envelopes
CHAPTER 5. CODE GENERATION

a number of machine instructions. The nodes in the graphs at the moment are those of the abstract machine. Looking at the largest of the graphs, it includes a \texttt{fsqr} which is unlikely to be a machine instruction. Such nodes can be viewed as a macro, or an abstraction and it must be expanded into more nodes, which implement a square root. (This may for example be a call to an operating system, or if a floating-point co-processor supports \texttt{fsqr} it will be a relevant co-processor instruction.)

It is occasionally necessary to create multi-way data-dependencies: compute something and ‘send’ it to two or more nodes, or ‘receive’ something from any number of nodes which are exclusive. Figure 5.5 demonstrates these two possibilities, and shows the T-code structure required to create them. The two instructions \texttt{rrw} (Read-Read-Write) and \texttt{wwr} (Write-Write-

![Figure 5.5: Multi-way data-dependencies](image)

Read) are all the node types necessary to implement a general data-dependency graph. \texttt{rrw} can duly be called a fork and \texttt{wwr} can duly be called an ‘accept’ operator. \texttt{wwr} accepts a piece of data on either of its input arcs and propagates it unchanged as its result.

5.2.5 Integrating control dependencies into a data dependency graph

Control dependencies are integrated into a data-flow environment. The mechanism is very simple, and is a type of node called \texttt{eval}. It may be used to return a value from the ‘middle’ of a graph.

Data dependencies are themselves an implicit type of control dependency. The graph is scheduled so that it can be executed only eagerly, and data-dependencies are the standard mechanism of describing functional parallelism — which is available for execution pipelining, etc. The first half of figure 5.6 shows the node \texttt{eval}. Dotted arrows represent control dependencies (whose arcs are uni-directional — which means that they can be represented on a machine with a single pointer); solid arrows represent data dependencies (whose arcs are bi-directional — which means that they must be represented by two pointers on a machine). The second half of the figure shows the \texttt{C} operator ‘?’, which performs a choice. It is similar to an \texttt{if}, except it returns a scalar value. It is quite easy to see that \texttt{eval} can introduce a control-dependency graph at any point in a data-dependency graph.
5.2. A MACHINE-INDEPENDENT GRAPH FOR TARGETTING

Function EVAL is the method of including control graphs inside data-dependency graphs. It triggers (s1), and returns the result received along (s2).

RESULT

This graph is equivalent to the C function

TEST ? IF_TRUE : IF_FALSE

and returns the value, according to TEST of either IF_TRUE or IF_FALSE (which are executed conditionally).

RESULT

This graph is equivalent to the F-code function

(\texttt{COMMA (L) (R)})

It evaluates (L), then evaluates (R), then returns the result of (R). (L) may be a subgraph which disposes of its result after evaluating it.

RESULT

This graph evaluates four sub-graphs (1) .. (4) in that order, and returns the value of the third one. The results of the other three are discarded. It is a more complex form of COMMA.

RESULT

Obviously, this graph is invalid. This graph evaluates four sub-graphs (1) .. (4), in that order, and returns the value of the third, or of the fourth one. (Which is indeterminate: it will [probably] return the value of the fourth).

It is necessary therefore for structures like this to only return the values of trees which are exclusive.

In the previous figure, the implementation of the C-function ? adheres to this restriction.

Figure 5.6: Control dependencies in data-dependency graphs

Figure 5.7: Control dependencies in data-dependency graphs
The use of eval to couple a data-dependency with a tree is shown in figure 5.7. The figure is in three parts, demonstrating the implementation of the F-code function comma. The function eval can thus be used to connect the result dependency to any part inside the eval's conceptual scope. The data-dependency must be connected to any number of sub-trees which are exclusive time-wise; the dependencies must have some sense, after all.

5.2.6 Traversing the graph

The graph as a whole, including data dependencies and control dependencies needs to be traversed in some way in order to perform a certain set of operations on it:

1. **Schedule** it. It needs to be scheduled (sequentialize it, pipeline it, etc.) In order to do so, a method of following only the control dependencies in the graph is required. This requires that the control dependency component of the graph is acyclic.

2. **Register allocation**, etc. Also a method of following the control dependencies in the graph from a writer to a reader, or from a reader to a writer needs to be provided. The dependency graph from any reader or writer is graph which may be cyclic; it may have any number of leaves, since the rrw and wwr nodes furcate the tree.

The full data-flow graph is regarded to be composed of a control graph, which is overlapped in places with data dependency graphs. Figure 5.8 demonstrates this idea. The two different graphs are constructed by removing all the eval nodes in the original data-flow graph, and recombining the components in the following way:

1. To create the control graph from the data-flow graph, for all of the eval nodes in all of its subtrees, the eval is removed and only its left dependency arc survives and this is connected to the respective top arc of the eval.

   This can be represented by the simple rewrite:
   
   \[
   y \leftarrow (\text{eval } d \ s1 \ s2) \leftarrow y \rightarrow s1
   \]

   Where \(\leftarrow\) is used to represent the control dependency, and \(\rightarrow\) represents the process of rewrite.

   The control graph constructed in this way is usually a set of binary nodes, some of which enforce a certain ordering of evaluation between their operands and others do not. There are many possible static schedules for this graph: the only constraint is that the operands of nodes are evaluated in advance.

2. To create the data dependency graph for a particular node in the tree (if it is the kind of node which permits data-dependencies — unlike seq, par, etc.) for all the eval nodes in all of its subtrees, the eval is removed and only its right dependency arc survives and this connected to the respective top arc of the eval.
5.2. A MACHINE-INDEPENDENT GRAPH FOR TARGETTING

Figure 5.8: The overlapping of data and control dependency graphs
This can be represented by the simple rewrite:

\[ y \leftrightarrow (eval \, d \, s_1 \, s_2) \leftrightarrow y \leftrightarrow s_2 \]

Where \( \leftrightarrow \) is used to represent the data dependency, and \( \rightarrow \) represents the process of rewrite.

Of course, when the compiler is implemented, these rewrites are not actually necessary. The two different graphs are 'constructed' by software by merely ignoring the unnecessary operands of \( eval \).

Data anti-dependencies are not explicitly represented. It is never necessary. The data-dependency graphs are logically trees. Where loops in the graph do occur, they are broken up with a node called \( val \). This is shown in figure 5.9. Structures like this are used to implement reduce operations, etc. The \( val \) is required to break the loop indicated by the dotted arrow.

There are a large number of cases where loops are created, and must be broken using \( val \) in order to make the scheduling procedure unambiguous, because it enforces the structure to be a tree: when a tree is scheduled, some recursive procedure must be applied to the tree; \( val \) stops the recursion carrying on indefinitely in certain cases where loops are created in a T-code tree. The \( val \) is transparent for data-dependencies, and blocking for control dependencies.

In figure 5.9 the data dependency graph is cyclic, but the control dependency graph is not. Since the data dependency graph is cyclic, if it is to be implemented on a register-based machine, the loop is implemented by a single register. On a three address machine, it can be correctly implemented by a piece of assembly language like

\[ adds \, r_2, r_1, r_1 ; \, r_1 = r_2 + r_1 \]

5.2.7 The representation of loops

In order to evaluate a data-parallel operation, the intermediate language includes a node which implements a loop. A data-parallel operation of more than one dimension is represented by a
5.2. A MACHINE-INDEPENDENT GRAPH FOR TARGETTING

This implements the loop:

\[
addr0[i] = (\text{real})addr1[i] + addr2[i], \quad 0 < i < \text{ext}
\]

adding an array of reals to an array of ints to produce an array of reals

Figure 5.10: Representing loops in T-code

The node responsible for loops is shown in figure 5.10. This diagram adds an array of integers to an array of reals to produce an array of reals. loop fires s0, s1 times; s2 is a tree of stride registers which are incremented for every loop. In figure 5.10, besides a counter which performs the loop from 0 to ext, are two stride (loop counter) registers which move in steps of sizeof_float, and sizeof_int respectively. cmb is just used to construct a tree, and does nothing but construct the tree. Referring back to section 4.3.1, this graph is implemented in the following way:

```c
for (ii=ce1=ce2=0; ii<ext; ii++, ce1+=sizeof_float, ce2+=sizeof_int) {
    [addr0+ce1] = (real)[addr1+ce2] + [addr2+ce1]
}
```

Where the notation \([ \ldots ]\) is used to represent a memory reference, or dereference. The addresses must be incremented in terms of the size of the basic types — which are sizeof_float and sizeof_int. The loop counter is called ii, and is implemented by the loop node of T-code. The two other loop counters ce1 and ce2 are defined in the cmb block of the loop. The implementation has a type coercion (fconv in the T-code graph).

In the graph, the node index is used to reference one of the loop counter variables. The node cmb is used to create a binary tree of loop counter variables. The loop counter variables
are T-code trees, which may be constants, or may be expressions. The three arcs of loop are the
block to execute, the number of times, and an arc to a tree of loop counter variables. Naturally,
index can only be used inside the scope of a loop.

5.3 Transforming F-code into a T-code graph

Since the compiler is to be partitioned at this particular point, transforming F-code into a T-
code graph is the main part of the code generator for the first partition. It is the task of the
second half of the partition to target an architecture-neutral graph onto a real machine.

The compiler infers a number of attributes on the main F-tree. These attributes (compiler
directives) are listed below:

1. A flag which marks the fact that static shape expressions (and hence extents) of an op-
eration are all constant. This is easily inferred during the stage of the compiler which
constructs static shape expressions.

2. A flag which marks whether the result of an operation is a scalar (either for each iteration
of a loop, or because the operation is scalar itself). Some nodes in the tree (those which
are enveloped in a lazy-evaluation scope) have rather than producing a block result, only
a scalar result (which is different for each iteration of the loop) since the conceptually
data-parallel computation occurs element-wise, sequentially. This attribute is described in
section 5.3.1.

3. Conversely, some operations really do need the ability to return block results. These are
implemented on a scalar machine, using a pointer to a block of memory — the pointer
is a scalar. This attribute is the opposite of the previous one (and is also described in
section 5.3.1). The result of a subtree which has been evaluated in a lazy manner is a
real data-parallel object. Since the compilation is to a scalar processor, a pointer to this
block-result is returned. Blocks are returned from sub-trees which are evaluated lazily ...
at the interface to the non-lazy environment.

4. Nodes in the F-code tree which require assignments are marked. Most run-time variables,
are single-assignment — they are assigned once, referenced once, then they are discarded.
Assignment in the data-flow graph is represented by a data-flow arc. (An assignment is
equivalent to putting a token on an arc.) This attribute bespeaks the position of data-
flow arcs and hence controls the construction of the data-flow graph. This is described in
section 5.3.3.

5. Positions of loops are marked. These loops are used to implement the data-parallel se-
manics of F-code. Also the positions of internal, loops such as those for assignments,
polynomial, reduce, etc. are marked. This attribute was discussed in the previous chap-
5.3. TRANSFORMING F-CODE INTO A T-CODE GRAPH

... ter. Its inference is a process of loop filtering. Along with the loop attribute is another which says the loop is an inner loop. (Formally an *inner-autonomous* loop.)

6. Positions of index manipulations. Positions where geometric operations take place. (Which are implemented on a scalar machine by computation on indices.) This attribute also controls the construction of the data-flow graph. This is described in section 5.3.5.

5.3.1 Lazy evaluation and lazy-environments

The implementation views F-code as a group of lazy-environments arranged in a tree which pass between them block-results (which are implemented by pointers to blocks of memory which are allocated on a number of stacks). Lazy-environments (*inner-autonomous* loops) are bounded by F-functions which cannot be implemented lazily, such as *create*, *mark*, etc.

This scheme is shown in figure 5.11, below:

![Lazy environments diagram]

The operand of this node is exactly the same as its result. It is the boundary of a lazy-environment, causing an *inner-autonomous* loop. Where there are more than one in a line, it was previously called a filter-group in loop-filtering.

The external arcs of a lazy-environment carry block-results (hence scalar pointers on a scalar processor). These arcs are labelled in the original F-tree with the block attribute — that is they carry a block result. Internally to a lazy-environment, the arcs are all scalars. Lazy-evaluation is the element-wise evaluation of a non-scalar object; effectively indices are parameters to a scalar function. Parts of the F-tree which cause *inner-autonomous* loops, pass a block result without modifying its value — such an F-function is *create*.

Considering a real F-program, figure 5.12 shows the program with lazy and eager sections marked.
((i1,i2) dyadic max ................................................. loop
  ((i1,i2) create A
   ((i3,i4) comma right ..................................... loop
     (assign .............................................. loop
       ((i5,i6) var name A)
       01 ((i6) ramp (const -5) (const 5) (const 1))
     )
     ((i3,i4) dyadic add
      ((i3,i4) var value A)
      00 (const 1)
     )
   )
   integer
   (const 11)
   (const 11)
  )
  00 (const 0)
)

Figure 5.12: Identifying block and scalar arcs
Figure 5.12 shows positions of internal loops (those for assignment, etc.) and the positions of loops inferred by loop-filtering which are responsible for implementing data-parallelism. The top Ω, the lazy environment, creates a matrix result element-wise referencing the result block element-wise of the Ω. The Ω itself creates and destroys an object, passing its main operand to the result (the block) without modifying it. The assignment executes in a lazy-environment of its own — this graph shows nothing about execution ordering: in fact the assignment occurs first of all, before any other environment in the graph. The right (comma) environment is also a lazy environment which adds the value of element-wise references to \( A \) to the scalar constant 1, producing a block.

Execution ordering of this set of environments is as follows:

1. As a side effect of the Ω, an object \( A \) is created.

2. The assignment environment is executed first writing a ramp across matrix \( A \).

3. The right (comma) environment is executed, adding 1 to every element of \( A \) to produce a result.

4. The Ω is executed, returning its operand unmodified.

5. As a side effect of the Ω, object \( A \) is destroyed.

6. The max environment is executed, returning a matrix.

The result of inferring block or scalar for every node of this tree becomes as the program below. The algorithm is reasonably straightforward, ascribing a block to pass the result of one lazy environment to another when the result passes via an Ω — via an F-function which limits the scope of lazy environments. When the result of a lazy environment is a scalar, this is passed as a scalar and not as a block. In the example, this F-function is the create. The scope is limited because this function has side-effects: it creates and destroys an object in memory.

```
([i1,i2] dyadic max ........................................... block, loop
 ([i1,i2] create A ........................................... block
  ([i3,i4] comma right ...................................... block, loop
   (assign .............................................. scalar, loop
    ([i5,i6] var name A) ......................... scalar
     01 ([i6] ramp (const -5) (const 5) (const 1)) ... scalar
    )
   ([i3,i4] dyadic add .............................. scalar
    ([i3,i4] var value A) ....................... scalar
     00 (const 1) ................................... scalar
   )
  )
 integer
  (const 11) ....................................... scalar
  (const 11) ....................................... scalar
 )
 00 (const 0) ....................................... scalar
)
```
5.3.2 Side-effects

Each node in the F-tree graph has up to three components:

1. Side-effect before
2. Its result
3. Side-effect after

(1) and (3) are not always present.

Three-point lazy code generation generates graphs for these components independently, and glues them together with this kind of glue-graph (which is a variant of COMMA).

Figure 5.13: Incorporating side effects

Figure 5.13 above depicts the interface to lazy evaluation environments. Lazy environments may have side-effects beforehand, and a side-effect after. For example, the F-code function create (which may surround a lazy environment — its primary operand) has the following:

1. The side-effect beforehand creates an object of a size specified by some of the create’s arguments

2. It goes on to evaluate a block or scalar result in a lazy environment. The result (this block, or scalar) is the result of the create function. Loops inside create are known as inner-autonomous.

3. Finally, it executes the second side-effect which destroys the object created beforehand.

This symbol can be represented by the T-code graph given in figure 5.13. The original F-code tree is traversed in the usual, recursive way, and three compiler functions are given for each type of node. These three compiler functions deal with:

1. Side-effects beforehand
2. The value component of the operation
3. Side-effects afterwards

Some F-functions do not generate side-effects, or only generate side-effects beforehand, or only afterwards. All F-functions produce a value component. There are thus a number of variations of glue-graph from the one shown in figure 5.13, however their purpose is always the same: to arrange sequential ordering and to pipe the value component out to the ‘top’.
### 5.3. Transforming F-code into a T-code Graph

<table>
<thead>
<tr>
<th>F-function</th>
<th>Side-effect before</th>
<th>Value component</th>
<th>Side-effect after</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST †</td>
<td>evaluate right &amp; hold</td>
<td>value</td>
<td>destroy value</td>
</tr>
<tr>
<td>HOLD</td>
<td>create object</td>
<td>evaluate operand</td>
<td>destroy object</td>
</tr>
<tr>
<td>TEMPLATE</td>
<td>create global</td>
<td>scalar pointer</td>
<td></td>
</tr>
<tr>
<td>CREATE</td>
<td>destroy global</td>
<td>scalar 0</td>
<td></td>
</tr>
<tr>
<td>VAR</td>
<td>create local</td>
<td>scalar pointer</td>
<td></td>
</tr>
<tr>
<td>SELECT †</td>
<td>record heap state</td>
<td>evaluate operand</td>
<td>restore heap state</td>
</tr>
<tr>
<td>GLOBAL</td>
<td></td>
<td>evaluate ramp</td>
<td></td>
</tr>
<tr>
<td>DISPOSE</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>LOCAL</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>MARK</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>RAMP †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>MONADIC †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>DYADIC †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>CHOICE †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>REDUCE †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>TRANS †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>SECT ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>SLICE ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>REPL ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>PACK ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>GATHER ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>DIAG †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>TRANSFORM †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>PART †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>COMP ‡</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>POL ‡ *</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>DISPLACE †</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td></td>
<td>evaluate operand</td>
<td></td>
</tr>
<tr>
<td>SHAPE</td>
<td>the assignment</td>
<td>evaluate operand</td>
<td>channel destroyed</td>
</tr>
<tr>
<td>ASSIGN *</td>
<td>channel created</td>
<td>evaluate operand</td>
<td>!channel</td>
</tr>
<tr>
<td>CHANNEL</td>
<td></td>
<td>evaluates operand</td>
<td></td>
</tr>
<tr>
<td>PUT</td>
<td></td>
<td>?channel</td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>sequentialization</td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td>the comma</td>
<td>not-discarded result</td>
<td></td>
</tr>
<tr>
<td>COMMA †</td>
<td></td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>PAR</td>
<td>parallelization</td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>the loop</td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>SPAWN *</td>
<td>the spawn</td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>the if</td>
<td>integer result</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>the call</td>
<td>scalar result</td>
<td></td>
</tr>
</tbody>
</table>

† — evaluated lazily. ‡ — requires an index manipulation. * — [may] require internal loops.

Table 5.1: F-functions and side-effects
Table 5.1 shows a list of F-functions, their side-effects beforehand, values returned, and side-effects afterwards. Thus for every node which is visited (on the return journey), the code generator does:

1. Generate T-code graph for 'before' side-effect
2. Generate T-code graph for 'after' side-effect
3. Generate value computation (in terms of nodes already visited)
4. If the node is marked to be the position of loops, generate loops (whose extents are given by the T-code graph generated for the respective F-code shape trees).
5. If the node is marked to need index manipulations, generate them
6. Glue the components together

Since index manipulations occur only for functions which are implemented lazily, 1 and 5 are exclusive. The T-code graph is associated with the appropriate node in the F-code tree, which allows 3 to work.

5.3.3 Scopes of single-assignment variables

Single-assignment variables are the anonymous variables in an F-code tree: not those variables with an identifier created explicitly by F-code functions hold, create, etc. Such F-functions which create named variables create and destroy them as side-effects.

Anonymous variables are single-assignment because F-code is represented by a tree: that is they are elementally written to once, and read at most once. Lazy-evaluation means that some or all elements of an object may never be read at all, at the interface between lazy environments caused by side-effecting functions of F-code. Side-effects limit the amount of laziness of the implementation.

Single-assignment arcs are always represented in T-code by data-flow arcs which are effectively sent to and received from; T-code inherits the fact that they are single-assignment.

There is approximately one single-assignment, perhaps non-scalar, variable required for each node of the F-code tree. Ω nodes may not require an assignment for their major operand because they do not modify it. Ω nodes are merely synchronization points.

Figure 5.14 shows single-assignment variables required to implement the example of section 5.3.1. (The positions of assignments denote positions of single-assignment variables.) These are internal (anonymous) variables, nothing to do with the variables declared by the F-code program, such as A.

If one considers the assignment, the single assignment variables do:

1. For the left hand side of the assignment, v1 = & A[i5, i6]. Some variable is the address of A referenced by i5 and i6, which are the indices of the assignment's lazy environment.
2. For the leaves of the right hand side of the assignment, $v2 = -5$; $v3 = 1$; assign the parameters of ramp to variables.

3. For the ramp on the right hand side of the assignment, it may be calculated by $v4 = v2 + (v3 \times i6)$.

The full assignment environment may be implemented using the C code:

```c
int assign_environment() {
    for (i5=0; i5<...; i5++) { for (i6=0; i6<...; i6++) {
        int *v1; int v2, v3, v4;
        v1 = &A[i5, i6];  /* Take the address of A[i5, i6] */
        v2 = -5; v3 = 1;  /* Parameters of the ramp */
        v4 = v2 + (v3 * i6);  /* Do the ramp */
        *v1 = v4;             /* Do elemental assignment */
    }
    return 0;             /* Assign returns 0 */
}
```
Positions of single-assignment variables in the full program are inferred in the following way:

- Functions which do not modify their major operand do not require an assignment. Such as `create` in the figure.
- Discarded operands of `comma` never need write their result anyway. In the figure, the `right` does not need a variable from the `assign`.
- `masks` are transparent: they have no longer any bearing on the compilation process. It has been shown that the process of shape-inference makes them defunct, since it distributes and names indices.
- All other nodes in the F-code tree require an assignment, if they take active part in the computation. In the figure, the second operand of `ramp` does not need to be assigned, nor does it every need to be evaluated, since the result of the `ramp` is defined to be

\[ r[i] = b + (i \times s) \]

where \( b \) is the base, and \( s \) is the step. The second argument (the `end`) needs to be evaluated if it is not constant, if one or more of the extents of its lazy environment depends on it.

The program is then:

```plaintext
([i1,i2] dyadic max ... block, asg, loop
 ([i1,i2] create A ... block
    ([i3,i4] comma right ... block, asg, loop
       (assign ... scalar, asg, loop
          ([i5,i6] var name A) ... scalar, asg
          01 ([i6] ramp ... scalar, asg
             (const -5) ... scalar, asg
             (const 5) ... scalar
             (const 1) ... scalar, asg
          )
       )
    )
    ([i3,i4] dyadic add ... scalar, asg
       ([i3,i4] var value A) ... scalar, asg
       00 (const 1) ... scalar, asg
    )
  )
 integer
 (const 11) ... scalar
 (const 11) ... scalar
  )
00 (const 0) ... scalar, asg
)```
5.3.4 Loops

Loops, by which is meant the loops instituted to implement data-parallel semantics, are tagged in the tree. These loops occur at the 'top' of lazy environments. They provide an index space for an element-wise evaluation of the lazy environment. Loops are represented in T-code, as was previously shown in this chapter, with common strides taken out into cmb blocks. The common-stride (cmb) blocks make sure that the minimum necessary number of indices are used to implement loops.

5.3.5 Indices and implementing geometric operations

Indices are arguments to lazy environments. Lazy environments evaluate a result element-wise, using the indices they are provided with. Geometric operations are then nothing more than arithmetic on indices. Every node in the original F-code tree has associated with it a number of indices (normally one for every of the function's dimensions). In section 4.4 in the previous chapter, a notation for defining active indices on a particular F-function was shown. This is used again here.

A number of the functions use internal indices to implement loops: these are not geometric operations — the functions are reduce, assign, gather, pol. Real geometric operations are repl, comp, sect, slice, transform, pack. It was said loc. cit. that the value of internal indices for these functions may depend upon the values of external indices. For some of these functions, the internal index is set to a particular value by one of the arguments of the F-function. For example, sect sets a particular internal index to a constant value. You will notice that diag and transp were implemented automatically by shape-inference: they require no time to evaluate, since they merely guide the distribution of indices: they are operations which construct shape trees only. transp's 'internal' indices are only the same as its external indices re-ordered. diag's 'internal' indices are the same as one of the external indices duplicated: a full diag on a box A is evaluated like A[i_1, i_2, i_3].

Internal loops are implemented in just the same way as standard loops which implement data-parallelism but they may occur inside a lazy environment — here internal loops are used to evaluate an element-wise result of a lazy environment.

All geometric operations are side-effects which occur before the evaluation of the function's main parameter. This is because they evaluate values for internal indices which are used by the parameter. Geometric operations are evaluated in the following ways:
sect: the internal index $i_N^i$ is set to the value of $\text{EXPR}.i$. The side-effect of the sect is the assignment (and hence the evaluation of $\text{EXPR}.i$) $\text{EXPR}.i$ is scalar and therefore is supplied with no external indices.

For example in

```
(sect 0
    ([i1] A)
    (const 13)
)
```

the geometric operation is implemented by $i_1 = 13$.

slice: the internal index $i_N^i$ is set to the scalar value of $\text{EXPR}.i$. In this case $\text{EXPR}.i$ is evaluated lazily with the external index $i_N$. Here the superfix $^s$ denotes parameter $\text{EXPR}.s$, while no superfix denotes the slice function itself.

For example in

```
([i2] slice 0
    ([i1] A)
    ([i2] B)
)
```

the geometric operation is implemented by $i_1 = B[i_2]$.

comp: if $e_N^s \neq \infty$, and an extra index is required, the index generated for the right hand side is assigned the external index $i_N$ minus the the appropriate extent of the left hand side $d_N$. The extra index is not generated if dimension $N$ is not an active dimension of the right hand side (in which case the extent $d_N$ of the right hand side, after expansion, is 1.)

For example in

```
([i1] comp 0
    ([i1] const "gaum")
    ([i2] const "less")
)
```

the geometric side-effect is implemented by $i_2 = i_1 - 4$, since the length of "gaum" is 4. It is only necessary to evaluate this side-effect when $i_1 > 4$ (when the right hand side is being evaluated).
transform: all of the external indices are relevant indices for associated members of \( \text{EXPR.t} \).

The rank of the result is the same as the number of \( \text{EXPR.t} \). The internal indices are assigned the externally indexed values of \( \text{EXPR.t} \).

This is a naive approach of implementing transform. There is a good deal of optimization which can be applied to this. Some elements of \( \text{EXPR.s} \) may be evaluated more than once (if it is duplicated by multiple references to it). This implementation is not as selective as it can be. For the first implementation of F-code, it remains as it is.

For example in

\[
([i_1,i_2] \text{transform}
  \begin{align*}
    ([i_3,i_4] & A) \\
    ([i_1,i_2] & B) \\
    ([i_1,i_2] & C)
  \end{align*}
\]

the geometric operation is implemented by \( i_3 = B[i_1,i_2] ; i_4 = C[i_1,i_2] \).

repl: In order to implement a repl, the mathematical function \( \text{mod} \) is used. The internal indices are calculated as the associated external index modulo the appropriate extent of \( d^t \).

For example if the result of a repl is the string “abcdabcdabcdabcdabcd” which was attained by replicating the string “abcd” five times:

\[
([i_2] \text{repl}
  \begin{align*}
    ([i_1] \text{const } & \text{"abcd"}) \\
    \text{(const 5)}
  \end{align*}
\]

the geometric operation (the side-effect) is implemented by \( i_1 = i_2 \mod 4 \). Some of the external indices may have no bearing on the object.

\[
([i_3,i_4,i_5] \text{repl 011}
  \begin{align*}
    ([i_1,i_2] & A) \\
    \text{(const 10)} \\
    \text{(const 10)} \\
    \text{(const 10)}
  \end{align*}
\]

In this example, \( i_1 = i_4 \mod 10 \) and \( i_2 = i_5 \mod 10 \). \( i_3 \) has no bearing upon any internal index \( (i_1,i_2) \). \( i_3 \) is only used as part of the address for storing part of the result. This function fleshes out a matrix into a box. The dimension which previously did not exist is flesned out to length 10 — each layer of which is the same.
CHAPTER 5. CODE GENERATION

pack : like repl, the _internal_ indices are evaluated from the _external_ indices. It is the most inefficient geometric operation, implementation-wise; fortunately it will be rarely used. The _external_ indices are first combined to provide a lexicographical number. The _internal_ indices are then generated from the lexicographical number.

For example in

```plaintext
([[i1,i2,i3] pack
  ([[i4,i5] A) <assume A is size [2,3]>
  (const 4)
  (const 6)
  (const 8)
)

from the _external_ indices, the lexicographical number L is i1+(i2*4)+(i3*4*6). Then, due to the size of A, i4 = L % 2 and i5 = (L / 2) % 3.

Each of these functions sect,slice,comp,transform, repl,pack requires a side-effect to be placed ahead of the evaluation of the main function. Each instance of these functions is marked with an attribute _index_, meaning index manipulation occurs here.

In the function

```plaintext
([[i6] sect 0 ........................................... index
  ([[i6,i5] slice 0 ................................... index
    ([[i4,i5] transform ................................ index
      ([[i2,i3] pack ................................... index
        ([i1] A) <assume length is 20>
        (const 10)
        (const 10)
      )
      01 ([[i5] B)
      ([[i4,i5] C)
    )
    ([i6] D)
  )
  (const 10)
)
```

the geometric operations are automatically evaluated by

```plaintext
i6 = 10; (sect)
i4 = D[i6]; (slice)
i2 = B[i5]; i3 = C[i4,i5]; (transform)
L = i2+(i3*10); i1 = L % 20; (pack)
```

5.3.6 Examples

Following is a number of concrete examples of the transformation between F-code and T-code.
Example 1

The first example is a very simple F-program which describes choice. Figure 5.15 demonstrates the creation of a section of T-code from an F-code tree. The program:

\[
\text{choice} \quad (\text{M}) \quad (\text{T}) \quad (\text{F})
\]

This program produces a vector result which is the choice of T and F, depending upon M.

When this graph is executed lazily (elementally), all of its arcs hold and yield scalar values. It is ‘called’ by an environment which supplies an index (i1), and receives a scalar result. The environment which call this graph elementally and compose a vector result in a block of memory elsewhere. (This part is not shown in the graph).

The choice is implemented using the T-code equivalent of the C ? operator, which was shown previously in figure 5.6. Which is a data version of if. The T-code graph is the equivalent to the pseudo-code:

```c
if (m[i1] == true) {
    x = t[i1];
} else {
    x = f[i1];
}
```

where x is the outward dependency.
Example 2

Figure 5.16 demonstrates the creation of a section of T-code from an F-code tree. The tree represents the F-code program section:

```
([[i1] dyadic max
  ([[i1] var value A)
  ([[i1] ramp
    (const 1)
    (const 21)
    (const 2)
  ])
)
```

The program requires an index (in the T-code graph, this index is referenced using the index nodes). The index is used in the computation of the ramp and secondly it is used as an index into variable A. The section of T-code tree given is a scalar graph, with effectively a parameter which is an index. It returns a scalar result.

Example 3

This example is a simple extension of the previous one. The tree represents the F-code program section:

```
(sect 0
  ([[i1] dyadic max
    ([[i1] var value A)
    ([[i1] ramp
      (const 1)
      (const 21)
      (const 2)
    )
  )
  (const 5)
)
```

which is the previous example enveloped with a sect. The compilation of an entire F-tree merely requires the tessellation of translated fragments. The result of this F-code tree, therefore, is scalar, which is the result of the conceptually vector result of the max with its index fixed at the value 5. This is shown in figure 5.17. This is the first example of an index-manipulation. The index is merely set to a constant value. It is an internal index, and also its value is not dependent on any external (to this F-code tree) index.

In the figure, DICP is the T-code function which allocates an entry in the dictionary. This entry consists only of a long word of memory, which is a pointer to data-objects allocated by create on the heap. TUPREF is an assembly-language level instruction which loads its operand (which is a tuple) into a register. Thus TUPLEF(DICP) loads the dictionary entry into a register, and is thereby the start address of a data object in memory which can be used as an operand to LD.
Figure 5.16: Example 2

Figure 5.17: Example 3
Example 4

Figure 5.18 is another example of index manipulation. This F-code tree composes two vectors together to produce a longer vector.

```
([i1]comp 0
  ([i1] const "abcdef")
  ([i2] const "ghijkl")
)
```

In the figure, the comp is implemented using an if. The value of index i1 is incoming. If it is less than 6, which is the extent of the left operand, the left operand returns its scalar result. Otherwise, i2 has to be evaluated (an index manipulation), such that \( i2 = i1 - 6 \), then the right operand returns its scalar result, which is indexed using i2.

![Figure 5.18: Example 4](image_url)

In this figure, the T-code function STRING has a parameter which is an index.
Example 5

Figure 5.19 is an example of reduce, which introduces an *internal* loop.

```
([i1] reduce add
  01 ([i1,i2] A)
)
```

In this program, (A) is an F-expression which returns a matrix result. One of the indices of A are used in the reduce, the other is not. This graph implementing this function returns elemental values of a vector result, where the vector is obtained by a reduce along one of the dimensions of a matrix. The operation of the reduce is add, thus it sums along one dimension.

The extents of A are assumed to be x1 and x2 respectively in the figure. This figure shows the structure of a reduce along a single dimension. Multi-way data-dependencies are simplified (there will be rrw and wwr on these arcs) for simplicity. The T-code graph A! is the graph produced for the F-code tree A. Valid indices inside it are both i1 and i2. i1 comes from the environment of the reduce, however — i2 is generated by the loop. The value of i2 is referenced in the standard way, using an index node (the arc leads to the loop node) — this is also shown in the diagram.
Example 6

This is another example of an internal loop, for the F-code function assign, which does a data-parallel assignment. The actual value component of an assignment is the integer value 0. The assignment is considered to be a side-effect of the assignment. This figure implements the

\[
\text{F-function}
\]

\[
\text{assign}
\]

\[
\text{(var name A)}
\]

\[
\text{(dyadic add}
\]

\[
\text{(const 0.5)}
\]

\[
\text{(var value B)}
\]

\[
\}
\]

where A and B are vectors, the minimum extent of which is ext. A is real, and B is integer (thus there is a type-coercion inserted during the type-inference stage — which is shown in the graph). The assignment is the side effect, and the value component is the T-code graph (mov (0)). The glue-graph does the assignment first, and then the value.
Example 7

This is an example of create. The side-effect which occurs before its main operand is evaluated is to create an object in memory. In F-code the primitive to allocate memory is alloc, and the object to destroy an object is free. T-code assumes a simple linear address-space.

\[
\text{(create } A \\
\quad (B) \\
\qquad \text{integer} \\
\qquad (\var \text{ value } B) \\
\qquad (\var \text{ value } C) \\
\)}
\]

where \(B\) is an F-expression which operates on the created variable. The synchronization point is created (shown in figure 5.13), since create has a side-effect occurring before the function and side-effect occurring afterwards. The side-effect beforehand allocates an object of size \(4 \times B \times C\), which is aligned to 4-byte boundaries, and writes the resulting handle to memory at the address of identifier ‘A’. The function goes on to evaluate \(B!\) and return this as the result, and finally, the memory which was allocated is destroyed by the free.

Inside \(B!\), the object is referenced using dicp, which is a dictionary entry for the object. (It is actually implemented as a global pointer which points to the block of memory).

This F-function can be implemented by the following piece of C code:

```c
static long *A=NULL;

    A = calloc(B*C,sizeof(long));
    ...
    free (A);
    ...
```

It is reasonably obvious that the graph is its equivalent.
Figure 5.21: Example 7
5.3. TRANSFORMING F-CODE INTO A T-CODE GRAPH

Each of these examples should have given a reasonable indication of the manner of constructing a T-code graph. Side-effects, etc. are glued together at the top of a representative T-code graph for each node of the F-code graph, in a bottom-up manner.

The only exception to the bottom-up manner of tree-construction is for indices. Referring back to figure 5.17, the arcs for i1 refer to something 'higher' up the T-code graph (higher up the F-code tree it was created from.) Implementation-wise these are a form of 'forward-reference'.

5.3.7 Blocks, memory and stacks

Each node of the primary F-code tree is marked to have a certain number of attributes which were introduced at the start of this section: considering the simple F-function

\[
\text{dyadic max}
\begin{align*}
0 & 1 \text{ (const "balls&")} \\
10 & \text{ (const "chains")}
\end{align*}
\]

is equivalent to

\[
A[i,j] = \text{max( "balls&"}[j] + "chains"[i]). \\
0 \leq i < 6, 0 \leq j < 6
\]

which can be implemented in C by:

```c
A = calloc(36,1);
for (j=0;j<6;j++)
    for (i=0;i<6;i++)
        *(A+i+j) = \text{max ( "balls"}[j], "chains"[i] );
```

Figure 5.22: Evaluating blocks
After the attributes are inferred for this function, it becomes

(dyadic max 01 (const "balls") 10 (const "chains") )

attributes are labelled <..>

The entire function returns a block result. It is is evaluated inside a loop since the max is attributed with a loop. As such, the T-code graph is given in figure 5.22. The return arc of this graph returns a pointer to a temporary block of memory — the fact that it is a pointer is noted in the F-code tree by the attribute block.

This block is available for assignment: consider the augmented F-function:

(hold A
 (B)
 (dyadic max
  01 (const "balls")
  10 (const "chains")
))

The right-hand side of the hold is identical to the previous graph, and it can simply be represented as in figure 5.23. If C in the graph is substituted for the F-function (dyadic max ...), it can be seen that it is evaluated in the order required for a hold operation. Secondly, the object is destroyed after the hold is terminated. In this graph, memory is consistent: the object is allocated and destroyed. Comparing figure 5.23 and figure 5.21, the two T-code graphs are
the same at the top, as they should be: hold is effectively a create, which creates an object with a pre-determined value.

It is necessary to make the memory-usage of all graphs consistent, such that there are the same number of allocs and frees created for internal variables, and creates; or there may be one less free if the entire F-code tree returns a non-scalar object. *Inner-autonomous* loops, as shown in figure 5.22 where an object is created create an object in memory, which must at some point in the future be destroyed. If this block is used as an operand to another lazy environment, the operand's memory must be destroyed afterwards.

```
(dyadic max <assign> <static> <block> <loop> [3]
  (create A <static> <block> [1]
    (dyadic add <assign> <static> <block> <loop> <inner> [2]
      (var value A) <assign> <static> <scalar> [2]
      0 (const 1) <assign> <static> <scalar> [2]
    )
    integer (const 10)
  )
  0 (const 0) <assign> <static> <scalar> [3]
)
```

This program is nonsensical, however it is used to demonstrate a point of the code generator because it is a simple program. The order of execution is indicated in square brackets. The create is executed first [1] (creating a variable), then the *inner-autonomous* loop is executed (numbered [2]). This returns a block result, and A is then destroyed. The block result is used as an operand to loop [3], which in turn returns a block result, and the block result for the *inner-autonomous* loop is then destroyed. This function is shown in figures 5.24 and 5.25. In figure 5.25, a few symbols are introduced to show the fact that the full graph is composed of smaller units in figure 5.25. Figure 5.25 shows the scope of an *inner-autonomous* temporary variable.
Evaluates the vector result of $\phi(\beta, \chi)$ element-wise and lazily, returning the result to $\alpha$ which is a block result.

The type-size is $\omega$ and the extent of the vector is $\psi$.

The result memory is allocated using the allocate function $\mu$.

Figure 5.24: Symbols for figure 5.25
5.3. TRANSFORMING F-CODE INTO A T-CODE GRAPH

In figure 5.25, the scope of the temporary variables is shown. The inner ‘temp’ loses its scope as the parameter of the max loop, and is then destroyed. The create causes an inner-autonomous loop, which is the add loop.

![Diagram of F-code to T-code transformation](image)

**Figure 5.25: Destroying inner temporaries**

Formerly it was shown that allocation of memory was done with the alloc function of T-code. The most efficient way of implementing internal variables of this form is on a stack, or number of stacks. One sees that this method of memory allocation for internal variables can be implemented using two stacks only, avoiding stack fragmentation.

An operation in the F-code tree assumes that all of its operands are on the same stack. The result is written on another stack, and its operands are freed; this way, since the use of stacks is interleaved, there is never stack fragmentation and thus never a need for garbage collection. The same is true for a T-code graph. The use of two graphs for internal variables is shown in figure 5.26. Two stacks are used. It is possible to guarantee that all the operands of a T-code function are on the same stack, because the control-flow component of a T-code graph is a tree. (The only prerequisite is that it is a tree.) Allocating an object on a stack is, of course, very fast — just requiring a single addition to move the stack pointer, and an additional register to hold a pointer to the position on the stack of the object.
In order to create objects for create and hold, etc. which are not internal variables, a third stack is needed. This is because the scopes of these variables are incongruous with the scope rules for internal variables. This (the heap) is permissibly a stack because the scope rules for it are defined by the F-code program, and it is a tree of scopes: objects are destroyed in exactly the opposite order as they are destroyed. However, this stack is incongruous with the two-stack required to implement internal variables.

Referring to table F.8 in appendix F, three different sets of memory allocation functions are given in T-code:

1. alloc,free:

   Following the scope rules of F-code, objects which are created and subsequently destroyed by the F-functions create and hold require a single stack of their own.

2. alios,fres:

   Secondly, following the scope rules of single-assignment internal variables, two stacks (a two-stack) are required to avoid memory fragmentation; and thereby garbage collection is not needed because they are simple stacks. This ideas was first seen, by me, in the implementation of E-code [EVA91]. These stacks are used to hold block results. The interleaving of stack use is shown in figure 5.26.

3. alloh,freh:

   Finally, for additional functions of F-code which allocate and destroy heap memory, a real heap is required. (These functions are global, dispose, local, mark, etc.)

All of them do essentially the same thing: they allocate a number of aligned blocks of memory (of a specified size, aligned to the same size.) The ascription of particular stacks, of the two-stack,
to instances of fres and allos cannot be done yet since the construction of the T-code graph proceeds bottom-up, and the ascription must occur top-down: the result of a particular T-code graph is on a particular stack, of the two-stack, and other subordinate lazy environments must have placed their results on a stack according to their depth, such that stack use is interleaved — see figure 5.26. (They are numbered top-down.)

The two-stack implements the anonymous variables of F-code passed between lazy environments. It operates as follows: the environments are computed bottom-first. Each environment is tagged with which stack it requires the result to be computed onto — and thereby knows that its operands must be computed onto the other stack. It arguments are pre-computed onto the appropriate stack, sequentially. It has pre-inferred the size of its arguments (in the shape inference stage of the compiler), and thus stack pointer relative offsets to operands are known. It computes a result onto the other stack, and destroys the operands to this environment. This result becomes an operand to another environment.

This has now given an overall view of the task of transforming F-trees into T-code graphs. At this point, the F-tree is discarded. Subsequent targetting is done on T-code graphs only. T-code, as it is, is sufficiently plastic to implement it congruently on pipelined RISC processors. Targetting is the subject of the next chapter.
Chapter 6

Targetting

- 6.1 — Transforming T-code into machine instructions ...................................................... 169
- 6.2 — Inference of addressing modes .................................................................................... 178
- 6.3 — Giving the graph a spine (sequentializing) ............................................................... 182
- 6.4 — Iterative scheduling and the current implementation ........................................... 184

6.1 Transforming T-code into machine instructions

With a T-code graph now produced, the next stage of the compiler is to transform it into congruent i860 code. This is the first part of the thesis which deals with anything which is architecture-dependent. The node types of the T-code graph have some correspondence with real machine instructions of RISC processors. The T-code graph is quite similar to a graph of real RISC instructions except the abstract machine is assumed to be very orthogonal which real machines seldom are.

Initially, the graph is rewritten simply using two basic types of rewrite called $\sigma$ and $\rho$ which are discussed shortly. These transform T-code into an initial, unscheduled, graph of i860 instructions. Low-level scheduling and optimization requires more fundamental types of rewrite to adapt the graph for congruent use of the cache, memory pipelines and arithmetic pipelines.

Only a simple targetter has been implemented, which does not make use of pipelining, but this chapter also describes a more congruent version which will be implemented to produce high code quality. The first coding of the targetter has been a mistake: the wrong language was used to implement it, but the techniques given in this chapter are still valid.

6.1.1 Primary rewrites $\rho$ and $\sigma$

The primary translation between abstract machine instructions and real machine instructions requires just two general rewrites called $\sigma$ which deletes a node from the graph, and $\rho$ which exchanges a node in the graph with an expansion. An expansion is another T-code graph which can replace a single node maintaining all of the existing arcs. These two rewrites are shown in
The two rewrites are $\sigma(t)$ and $\rho(t, U)$. In both cases, $t$ is a node in the graph which is a tuple such that $t = (x, d, s_1, s_2)$ where $d, s_1, s_2$ are arcs, and $x$ is an enumeration describing the node type. Where an arc is omitted (an empty graph), the symbol used is $\bot$. $U$ is a graph which supplants the original node $t$, as in the figure.

![T}he σ rewrite](image1)

![The ρ rewrite](image2)

Figure 6.1: Primary transformation rewrites

There are eight permutations for nodes, disregarding the few exceptions which are used to represent numerical constants, or labels, etc.

1. $t = (x, \bot, \bot, \bot)$ (a black hole, or sink)
2. $t = (x, \bot, \bot, s_2)$ (a unary node, without a return path) – representative of unary control dependency nodes
3. $t = (x, \bot, s_1, \bot)$ (never used)
4. $t = (x, \bot, s_1, s_2)$ (a binary node, without a return path) – representative of binary control dependency nodes
5. $t = (x, d, \bot, \bot)$ (never used)
6. $t = (x, d, \bot, s_2)$ (a unary node) – representative of monadic data-flow operations
7. $t = (x, d, s_1, \bot)$ (never used)
8. $t = (x, d, s_1, s_2)$ (a binary node) – representative of dyadic data-flow operations, loads, stores, etc. Occasionally, $d$ is called something else like $s_0$. Implementation-wise it requires the same structure.

Elemental selections from tuple $t$ are then $x_t, d_t, s_{1t}, s_{2t}$. For every node type of the abstract machine, a rewrite (a simple mapping) is given between the abstract machine instruction and real machine instruction(s).
6.1. TRANSFORMING T-CODE INTO MACHINE INSTRUCTIONS

In the following description, node and instruction are taken to mean the same thing. Some abstract machine nodes are unnecessary in the machine graph — for example sign extension nodes of the abstract machine are unnecessary on an i860, since it provides automatic sign-extension on loading a byte from memory. When the nodes are unnecessary, the rewrite used is $\alpha$. Otherwise, the abstract machine instruction is implemented using one or more machine instructions, the rewrite used is $\rho$.

Appendix G lists all the primary rewrites required to transform an abstract machine graph into an i860 equivalent. The mapping is one way, and is thus guaranteed to terminate. It is this set of rewrites which makes the abstract machine abstract, and this is the initial stage of the targeting transformation.

By way of example, $(\text{muls}, d, s1, s2) \xrightarrow{\omega} (\text{fxfr}, d, \bot, (\text{fmlow}, \ldots, (\text{ixfr}, \ldots, \bot, s1), (\text{ixfr}, \ldots, \bot, s2)))$

$muls$, the abstract machine instruction for integer multiplication has no direct equivalent on the i860 — instead, both operands (assumed to be in registers) are moved to the floating point register file, using $\text{ixfr}$ and an $\text{fmlow}$ (floating point multiply) completes the multiply, and the result in a floating point register is moved back to an integer register using $\text{fxfr}$. The registers are not allocated at this stage, and register file overflows (when no register is available) need to be handled with separate rewrite-transformations. The i860 assembler equivalent is thus (where the registers have assumed numbers).

\[
\begin{align*}
\text{ixfr} & \quad r1,f2 \\
\text{ixfr} & \quad r2,f4 \\
f\text{mlow} & \quad f2,f4,f6 \\
f\text{fxfr} & \quad f6,r3
\end{align*}
\]

move integer register to floating point register
move integer register to floating point register
floating point multiply low
move floating point register to integer register

The rewrites are defined in the following way: The symbol $\ldots$ is an arc pointing to the parent node, such that if tuple $t = (x_0, d, (x_1, \ldots, s1_1, s2_1), (x_2, \ldots, s1_2, s2_2))$, both occurrences of $\ldots$ are textually replaced by the arc leading to $t$. It is most easily described like this, without the use of labels. It is unambiguous.

Another example is:

\[(\text{odd}, d, \bot, s2) \xrightarrow{\omega} (\text{and}, d, \text{NUM}(1), s2)\]

which implements the function odd which produces a logical result. Its implementation assumes a certain numerical system — two's complement, and logical true is the value 1. Its equivalent i860 assembly language is:

\[
\begin{align*}
; (r2) = \text{odd}(r1) \\
\text{and} & \quad 0x0001,r1,r2 ; \text{take lowest-order bit}
\end{align*}
\]

Since the i860 is capable of calculating real results in single or double precision, in some cases two rewrites are given. A flag given to the i860 targetter is whether to use single or double precision for reals. Such rewrites are labelled $\omega_{\text{single}}$ and $\omega_{\text{double}}$. 
The i860 does not fully support the type complex. In order to support complex, a complex is held in a pair of registers (in single-precision mode), or a pair of pairs of registers (in double-precision mode). Each arc which carries a complex result is split into two arcs one of which carries the real component, the other carries the imaginary component. The node type called pair,

![Diagram showing the representation of complex numbers with pair, s/pairl, and s/pairh nodes.](image)

composes a pair from its component registers or pairs of registers. And spairl, spairh, dpairl and dpairh select a component of a pair, in either of the precisions.

A macro called `COMPLEX` is defined to be

\[
\text{COMPLEX}(t) \begin{cases} 
\text{double} & \begin{cases} (\text{dpairl}, \ldots, \perp, t), (\text{dpairh}, \ldots, \perp, t)), & x_t \neq \text{pair} \\
(s_1, s_2), & x_t = \text{pair} \end{cases} \\
\text{single} & \begin{cases} (\text{spairl}, \ldots, \perp, t), (\text{spairh}, \ldots, \perp, t)), & x_t \neq \text{pair} \\
(s_1, s_2), & x_t = \text{pair} \end{cases}
\end{cases}
\]

which splits a complex arc into a pair of arcs. If the complex arc leads to a node of type pair (which composes a pair of arcs), the pair is removed.

Then some examples of rewrites applied to complex operations are given below:

\[
(cadd, d, s1, s2) \rightarrow_t \\
(z1re, z1im) = \text{COMPLEX}(s1) \\
(z2re, z2im) = \text{COMPLEX}(s2) \\
t = (\text{pair}, d, (fadd, \ldots, z1re, z2re), (fadd, \ldots, z2re, z2im))
\]

which implements a complex-addition by performing two scalar floating point additions. The complex operands are first split using `COMPLEX`, and then paired to produce a complex result by using the pair.
which performs a complex conjugation by negating the imaginary component of its operand.

6.1.2 Transforming loops

Another rewrite, which can be classed as a kind of \( \rho \) rewrite, is that necessary for \texttt{loop}. Referring back to figure 5.10, the rewrite applied to loops first renames all instances of \texttt{index} to \texttt{val} (using a simple \( \rho \) rewrite), where \texttt{val} is a normal reference to a node (producing its value).

A loop has three main stages:

1. The \textit{pre-amble} — before the main loop is (a) the pipeline setup stage (b) initialization of loop invariants (addresses, etc.) (c) zeroing of loop indices

2. The \textit{loop body} — the main part of the loop

3. The \textit{post-amble} — clearing the pipeline
Figure 6.3 shows the expansion for a loop node. It reads approximately left to right. First is the pre-amble, which evaluates the extent \((ext)\) of the loop, and then sets an index variable to zero. The dependencies for this index are marked using a thick line. One can consider arcs like this as registers. Data flows to it and from it, gated at certain times: the pre-amble writes to it, the adds references it and adds on to it, and val references it. (Since this arc is multi-way, there will also be \(rrw\) or \(wrr\) nodes on its path, but these have been omitted for clarity.)

The loop body is executed repeatedly — the if checks when to terminate the loop. The adds repeatedly adds one to the index variable. The eq checks whether the index variable has become the same as the extent, at which point the other arc of the if is chosen, and execution enters the post-amble. The post-amble includes a stretch, which does nothing more than stretch the data-dependency for the index across into the post-amble. This is solely for register allocation purposes, since a register must be allocated throughout the loop. It is also necessary to stretch the register allocated for the extent across into the post-amble. The equivalent i860 for this graph is:

```
// pre-amble begin
or  r0,r0,r4  // set index to 0
...  // evaluate extent into r11
// pre-amble end

14:
// loop body begin
...  // do the main loop
// loop body end
adds 1,r4,r4  // increment the index
xor  r4,r11,r0  // compare with extent
bnc  14

// post-amble begin
// stretch (r4)  // allocate r4 up until this point
// stretch (r11)  // allocate r11 up until this point
// post-amble end
```

In this expansion, there have been no other loop counter variables required — just the index. (The cmb half of the loop was empty). In order to implement any number of other loop counters, additional \(\text{movs}\) are required in the pre-amble, additional \(\text{adds}\)'s to increment their values, and additional stretches are required to extend their scope. For example:

```
// pre-amble begin
or  r0,r0,r4  // set index to 0
or  r0,r0,r5  // set loop counter to 0
or  r0,r0,r6  // set loop counter to 0
...  // evaluate extent into r11
// pre-amble end

14:
// loop body begin
...  // do the main loop
// loop body end
adds 8,r6,r6  // step in size of complex
adds 4,r5,r5  // step in size of real
```
6.1. TRANSFORMING T-CODE INTO MACHINE INSTRUCTIONS

add r4, r4
xor r4, r11, r0
bnc l4

// post-amble begin
// stretch (r4)
// stretch (r11)
// post-amble end

This introduces two more loop counter variables, which have assumed registers r5 and r6.

6.1.3 Transforming if's

Similarly to loops, a $p$ rewrite is provided for if. if is a very convenient macro to use, but of course it is never a machine instruction. Figure 6.4 shows this rewrite. The expansion of if is context-sensitive. It depends upon the direction of the condition code (either set or clear, corresponding to true or false) which is operand $s0$ of the if. If, when the condition code is set, this corresponds to true, $\lambda$ is bc. Otherwise, if when the condition code is clear, this corresponds to true, $\lambda$ is bnc. This is explained further in section 6.1.4.

6.1.4 Condition codes

Arithmetic and logical comparisons, equalities and inequalities produce in the abstract machine a logical result. The i860 is one of a class of processors which puts the result of comparison not in a general register, but in a condition code flag. Associated with this flag are two conditional branch instructions — bc and bnc which branch when the condition code flag is set and clear respectively.
There are two contexts in which these logical operators occur:

1. The first requires the result to be stored in a register — if the logical operator is in an expression, and is not the root. For example in \((\text{or } (\text{and } A \ B) \ (\text{and } C \ D))\), the results of the two ands need to be stored in registers, and a logical or performs the root or. A logical or is implemented lazily in the conventional way: \(c = \text{or } \text{true} : r\) This evaluates the left hand side of the or — if this is \text{true} evaluating the right hand side of the or is unnecessary.

Logical and can likewise be implemented lazily in the conventional way: \(c = \text{and } r : \text{false}\) This evaluates the left hand side of the and — if this is \text{false} evaluating the right hand side of the and is unnecessary.

Transferring a result from the condition code flag to a register can be done in two clock cycles on the i860 using the inline expansion:

\[
\begin{align*}
\text{bc.t 17} \\
\text{mov true,r1} & \quad // \text{in the branch-delay slot} \\
\text{mov false,r1} \\
\end{align*}
\]

which moves true into assumed register r1, or false into r1 depending upon the condition code flag.

2. The condition for conditional branches. If the result is not needed for part of the computation, but instead just setting the condition code flag will do. For example in \(\text{if } (a = b)\), the result of \(a = b\) is not needed (it is for effect only, which is to set or clear the condition code flag).

The code generated (hence the graph rewrite) should be different in these two cases for efficiency. The abstract machine makes no differentiation between logical or, or bit-wise or, nor does it specify its context. Inferring the context is no problem at all, merely by tracing the data-dependency of the result of the logical operation. In the first case above, the result will be read somewhere, since it is used inside an expression. However in the second case, there will not be a data-dependency — since the result is not needed. The data-dependency is omitted in the \(\rho\) rewrite for if — see figure 6.4 — where s0 in the original abstract machine is a bi-directional data-dependency, but after the \(\rho\) rewrite is only a one-way control dependency. The missing direction indicates that the logical operation is for effect only.

A logical expression tree can be a mixture of operations for their value, and operations for effect (those which modify the condition code flag). The i860 includes the floating point instruction only for floating-point greater than (and floating-point less than or equal, which is identical); the other floating point comparisons must be constructed from it:
6.1. TRANSFORMING T-CODE INTO MACHINE INSTRUCTIONS

\[(\text{eq}, d, s1, s2) \rightarrow (\text{peq}, d, s1, s2) \quad \text{CC} \leftarrow (\text{src1} == \text{src2})\]

\[\text{ (\text{ne}, d, s1, s2) } \rightarrow (\text{pne}, d, s1, s2) \quad \text{CC} \leftarrow (\text{src1} == \text{src2})\]

\[\text{ (\text{gt}, d, s1, s2) } \rightarrow (\text{pgt}, d, s1, s2) \quad \text{CC} \leftarrow (\text{src1} > \text{src2})\]

\[\text{ (\text{lt}, d, s1, s2) } \rightarrow (\text{plt}, d, s1, s2) \quad \text{CC} \leftarrow NOT(\text{src1} \leq \text{src2})\]

The direction of CC for those marked with † are opposite to that required (ie. true relates to carry clear).

In order to implement \text{fge} and \text{flt}, the intermediate rewrite requires an or, and an and respectively.

\[\text{ (fge, d, s1, s2) } \rightarrow (\text{or}, d, (\text{gt}, d, s1, s2), (\text{eq}, d, s1, s2))\]

\[\text{ (flt, d, s1, s2) } \rightarrow (\text{and}, d, (\text{lt}, d, s1, s2), (\text{ne}, d, s1, s2))\]

Since the operands of \text{flt} are of the wrong direction, this is implemented instead, to produce the right direction, by De Morgan's theory:

\[\text{ (flt, d, s1, s2) } \rightarrow (\text{or}, d, (\text{lt}, d, s1, s2), (\text{ne}, d, s1, s2))\]

Similar optimizations exist by De Morgan's theory, when or and and are compiled lazily, but when negations really are required, of the effect-flag, the instruction pair:

```
bc.t 17
mov 1,r1 // in the branch-delay slot
mov 0,r1
```

are used, which is the same pair of instructions that moves the condition flag CC into a register. If this instruction pair is denoted by \text{INV}(\ldots), the alternatives of and and or compiled lazily are as below:

\[\text{ (or, d, s1, s2) } \rightarrow (\text{or}, d, s1, s2)\]

\[\text{ (or, d, s1, s2) } \rightarrow (\text{or}, d, \text{INV}(s2))\]

\[\text{ (or, d, s1, s2) } \rightarrow (\text{or}, d, \text{INV}(s1), s2)\]

\[\text{ (or, d, s1, s2) } \rightarrow (\text{and}, d, s1, s2)\]

\[\text{ (and, d, s1, s2) } \rightarrow (\text{and}, d, \text{INV}(s1), s2)\]

The † attribute percolates itself upwards. When the result of the and or or operation is required, not just for effect (not as the test operand of an it), it is transferred into a register using \text{INV}(\ldots).
Referring back to the $\rho$ expansion for $\textbf{if}$, the $\lambda$ parameter of the rewrite is another optimization. The test operand of the $\textbf{if}$ is for effect; if the direction of this is $\uparrow$, the $\textbf{bnc}$ alternative is chosen. This means that an inversion can be omitted.

Note: these $\textbf{or/and}$ instructions are opposed to the $\textbf{ior/land}$ instructions for integer arithmetic which cannot be executed lazily (they are bit-wise operations). This differentiation is made by the compiler to T-code (which is easy, just checking the type inferred for the F-code operation).

### 6.1.5 External calls and register coalescing

A $\rho$ rewrite which rewrites an abstract machine instruction into i860 instructions is effectively a macro which is expanded in-line. Since the abstract machine wields the full functionality of F-code, including all of the intrinsic mathematical functions of F-code, some of the functions, if expanded, would result in duplicated sections of tree. (Every instance of $\ln$, for example, would be expanded.) In such cases, it is useful to call a standard library, rather than have the code in-line.

The external call is a $\rho$ rewrite in just the same way — the arcs which are its interfaces exist before and after the rewrite. External calls require the code generated to conform to the calling-conventions for standard libraries, which are not necessary for in-line expansion:

1. they require active registers to be saved on the stack before the call & these registers to be restored after the call

2. arguments and results of the call must be placed in the right registers

Register allocation is not yet known; thus nodes marking the positions of register saves and restores are placed in the graph called $\texttt{savereg}$ and $\texttt{rstrreg}$ respectively, which are expanded only after register allocation has taken place.

A T-code node may have at most two arguments and at most one result; the arguments and results may be of any type, including complex. Since there are only a few, they can be passed to, and the result received from, library functions in registers. In order to conform to register-usage, nodes $\texttt{iccoalesce}$ and $\texttt{fcoalesce}$ are used (whose parameter is a register number).

### 6.2 Inference of addressing modes

After the rewrite to 'machine' instructions a graph remains with all of the appropriate i860 instructions named. However, as yet they do not have addressing modes, and the machine is still assumed to be orthogonal.

The i860 is a RISC machine and hence there are relatively few addressing Modes. Most operations are register-to-register based, except explicit load and store instructions. The addressing modes of the i860 are listed in table 6.1.
6.2. INFERENCE OF ADDRESSING MODES

Since the operands of the ADDS are (N, I), its addressing mode must be A_NII.

Since the operands of the ADDS are (I, I), its addressing mode must be A_III.

Since the operands of the FLD are (N, I), its addressing mode must be A_NIF.

In single-precision mode, this is the equivalent of

\[ \text{orf} \quad h\%11, r0, rX \]
\[ \text{fld}1 \quad s\%11(rX), fY \]

This is a single-precision floating-point to integer conversion:

\[ \text{ftrunc} . sd fX, fY \]
\[ \text{fxfr} \quad fY, rZ \]

(fY is a pair of registers, the lowest of which is selected by SPAIRL)

.data
.align 4
11: .byte 63,0,0,0 // 0.5

Figure 6.5: The inference of addressing modes
Some nodes in the i860 graph have a definite addressing mode, without inference. Others require the data-dependency component of the graph to be traversed from the reader to the writer in order to differentiate between possibilities: this is shown in figure 6.5.

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Explanation</th>
<th>Type of instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. III</td>
<td>isrc1,isrc2,idest</td>
<td>adds, subs, xor, and, or ...</td>
</tr>
<tr>
<td>A. NI</td>
<td>#consta,isrc2,idest</td>
<td>adds, subs, xor, or, orh ...</td>
</tr>
<tr>
<td>A. L</td>
<td>label</td>
<td>br, bnc, bc ...</td>
</tr>
<tr>
<td>A. IIS</td>
<td>isrc1,isrc2,shroff</td>
<td>bte, btne, ...</td>
</tr>
<tr>
<td>A. I</td>
<td>isrc1</td>
<td>bri, calli ...</td>
</tr>
<tr>
<td>A. NIS</td>
<td>#sconst,isrc2,shroff</td>
<td>bte, btne ...</td>
</tr>
<tr>
<td>A. FFP</td>
<td>isrc1,isrc2,fdest</td>
<td>fadd, fmul, l2as1, m12asm ...</td>
</tr>
<tr>
<td>A. FP</td>
<td>fsrc1,fdest</td>
<td>fix, ftrunc, fsetq, fmov ...</td>
</tr>
<tr>
<td>A. IFI</td>
<td>isrc1(isrc2),fdest</td>
<td>fld ...</td>
</tr>
<tr>
<td>A. NI F</td>
<td>#const(isrc2),fdest</td>
<td>fld ...</td>
</tr>
<tr>
<td>A. IPF</td>
<td>isrc1(isrc2)++,fdest</td>
<td>fld ...</td>
</tr>
<tr>
<td>A. NIPF</td>
<td>#const(isrc2)++,fdest</td>
<td>fld ...</td>
</tr>
<tr>
<td>A. Ni</td>
<td>#const(isrc2)</td>
<td>flush ...</td>
</tr>
<tr>
<td>A. NIP</td>
<td>#const(isrc2)++</td>
<td>flush ...</td>
</tr>
<tr>
<td>A. I</td>
<td>#const(isrc2)</td>
<td>nop, fnop, intovr, lock ...</td>
</tr>
<tr>
<td>A. FII</td>
<td>freg,isrc1(isrc2)</td>
<td>fst ...</td>
</tr>
<tr>
<td>A. FNI</td>
<td>freg,#const(isrc2)</td>
<td>fst ...</td>
</tr>
<tr>
<td>A. FIP</td>
<td>freg,isrc1(isrc2)++</td>
<td>fst ...</td>
</tr>
<tr>
<td>A. FNP</td>
<td>freg,#const(isrc2)++</td>
<td>fst ...</td>
</tr>
<tr>
<td>A. CI</td>
<td>ctrlreg,idest</td>
<td>ld.c ...</td>
</tr>
<tr>
<td>A. II</td>
<td>isrc1(isrc2),idest</td>
<td>ld ...</td>
</tr>
<tr>
<td>A. NI</td>
<td>#const(isrc2),idest</td>
<td>ld ...</td>
</tr>
<tr>
<td>A. II</td>
<td>isrc1,idest</td>
<td>mov ...</td>
</tr>
<tr>
<td>A. IC</td>
<td>isrc1,ctrlreg</td>
<td>st.c ...</td>
</tr>
<tr>
<td>A. INI</td>
<td>isrc0,#const(isrc2)</td>
<td>st ...</td>
</tr>
<tr>
<td>A. FI</td>
<td>fsrc1,idest</td>
<td>fxfr ...</td>
</tr>
<tr>
<td>A. IF</td>
<td>isrc1,fdest</td>
<td>ixfr ...</td>
</tr>
</tbody>
</table>

Table 6.1: Addressing modes of the i860

The addressing modes which are inferred are attributed to the node in the i860 graph. The inference occurs in a straightforward recursive pass of the control component of the graph. The flags involved in figure 6.5 are automatically assigned when the T-code graph is transformed to machine instructions.

i860 instructions along with requiring the addressing mode they are to use also require size flags. These flags indicate precision (either single or double) for floating point operations, and sizes for load/store instructions. It is possible for floating point loads for instance to load units of word length (using .l), of double word length (using .d) which is the size of double precision floats, and of quadruple word length (using .q) which is the size of two double precision floats.

For example, the \( p \) rewrite for \( \text{iconv} \) has two alternatives (this is from appendix G):

\[
(\text{iconv}, d, \perp, s2) \xrightarrow{\text{double}} t \\
(x^l, x^h) = \text{SPLIT}((\text{ftrunc}(.dd), \ldots, \perp, s2))
\]

\[
t = (fxfr, d, \ldots, x^l)
\]
6.2. INFERENCE OF ADDRESSING MODES

\[ (x^l, x^h) = \text{SPLIT}((f\text{trunc}(\text{sd}), \ldots, \bot, s2)) \]

\[ t = (f\text{xfr}, d, \ldots, x^l) \]

where

\[ \text{SPLIT}(t) \overset{\rho}{\mapsto} ((\text{spairl}, \bot, \bot, t), (\text{spairh}, \bot, \bot, t)), t \neq \text{pair} \]

The only difference between the implementations is the type of \( \text{ftrunc} \) used. In the double-precision case, the operand of the \( \text{iconv} \) is already double precision, and the flag is .\text{dd}, whereas in single-precision, the operand of the \( \text{iconv} \) is single precision, and the flag is .\text{sd}.

### 6.2.1 Immediate constants

There are no such things as immediate floating point constants, except floating point registers \( f0 \) and \( f1 \) always contain the single/double precision value 0.0. All other floating point constants must be loaded from memory, using the pair of instructions:

\[
\begin{align*}
\text{orh} &\quad \text{ha}_1, r0, rX \\
\text{fld.l} &\quad 1\text{ha}_1(rX), fY
\end{align*}
\]

which loads a single-precision floating point constant from an address labelled as \( 11 \) into floating point register \( fY \). It is assumed by the compiler creating the T-code graph that no machine allows immediate floating point constants: loading floating point constants is already a load instruction of the form \((\text{fld}, \ldots, (\text{tuploref}, \ldots, u), (\text{tuphiref}, \ldots, u))\), where \( u \) is the floating point constant. This is automatically rewritten, using a \( \rho \) rewrite, in the case of the i860 in single-precision mode, to \((\text{fld}, \ldots, (\text{lref}, \ldots, u), (\text{orh}, \ldots, (\text{tuploref}, \ldots, u)), \text{RZERO})\), which produces the instruction pair as above. If the machine really does allow immediate floating point constants, the \( \sigma \) rewrite is simply: \((\text{fld}, \ldots, (\text{tuploref}, \ldots, u), (\text{tuphiref}, \ldots, u)) \xrightarrow{\sigma} u\), deleting the load instruction.

Immediate integer constants which are used inside any instruction set are of a limited length, less than the real word length of the machine. Such constants in the i860 instruction set are limited to 16 bits; there are pairs of instructions to modify the top and bottom 16 bits of a register. In order to load a thirty-two bit constant into a register, the instruction pair is:

\[
\begin{align*}
\text{or} &\quad \text{low}_{16}, r0, rX \\
\text{orh} &\quad \text{high}_{16}, rX, rY
\end{align*}
\]

which loads a thirty two bit constant into \( rY \). The abstract machine assumes that any constant is a permissible immediate constant for instructions. Thus, all immediate constants in the i860 tree must undergo a range check. The \( (\rho) \) rewrite, when the range check fails is:
(int, d, x) \mapsto (orh, d, NUM(HIGH(x)), (or, .., NUM(LOW(x)), RZERO))

where

\[
\begin{align*}
NUM(n) &\models (int, .., (n)) \\
LOW(x) &\models (x&0xffff) \\
HIGH(x) &\models ((x << 16)&0xffff) \\
RZERO &\models (rzero, .., l, l)
\end{align*}
\]

### 6.2.2 Unavailable addressing mode

One addressing mode which is implied by the graph does not exist in the instruction set of the i860. The addressing mode would be called A_III — isrc0, isrc1(isrc2), for integer store instructions.

The nearest equivalent is A_NIi — isrc0, #const(isrc2). A rewrite has to occur for nodes which require this unavailable addressing mode, such that:

\[
\begin{align*}
(st, isrc0, RZERO, isrc2) &\rightarrow (st, isrc0, NUM(0), isrc2) \\
(st, isrc0, isrc1, isrc2) &\rightarrow (st, isrc0, NUM(0), (addu, .., isrc1, isrc2)) , isrc1 \neq RZERO
\end{align*}
\]

### 6.3 Giving the graph a spine (sequentializing)

The next part of the compilation process is transform the graph into a graph with a spine. This involves traversing the control component of the graph, and stringing together a spine of instructions, corresponding to the default sequential ordering of the instructions. This results in a more recognizable scheduling graph for register-allocation, etc.

An example of this transformation is given in figure 6.6.

The original abstract machine instruction (max rX, rA, rB), corresponding to the F-code function (dyadic max ...), is expanded into the T-code graph. The transformation creates a spine (the dark dotted line in the figure), and additional removes all eval, seq, par, val nodes because they are now unnecessary.

The spine gives a real sequential ordering which is:

```
subs rA,rB,r0
bnc 11
or rA,r0,rX
b 12
nop
11:
or rB,r0,rX
12:
```
This transformation destroys all control dependencies but the spine. The transformation is the low-level scheduling of the graph. Since there now exists a fixed sequential order, it is possible to perform code-compaction, register-allocation, etc. During the sequentialization process is is also possible to include peephole optimization.

Figure 6.6 shows the simplest possible schedule (the default schedule) of the instructions. It is an area of future work to perform pipelining; also to perform percolation scheduling to utilize the superscalar architecture of the i860. Of course, the order of sequentialization affects the number of live registers which are needed when sequentializing a more complicated graph.

For example, in order to avoid pipeline bubbles the schedule might be:

```
subs rA,rB,r0
bnc.t 11
or rB,r0,rX
or rA,r0,rX
```

```
11:
```

This schedule utilizes the branch-delay slots of the execution pipeline. It has also been optimized by a peephole optimizer.
6.4 Iterative scheduling and the current implementation

The current implementation allocates registers and outputs the default schedule of the code. This section describes a projected implementation, in comparison to the current one.

6.4.1 Execution pipelining

Execution pipelining is deceptively straightforward. In order to pipeline, all that is necessary is to modify the algorithm which links together the spine to take into account execution pipeline bubbles. That much is easy, but execution pipelining may only be done after register allocation. Register allocation is discussed shortly.

6.4.2 Arithmetic pipelining

A compliant model of the i860 arithmetic units needs to be implemented in order to make use of arithmetic pipelining. The approach to identify strict computational patterns at the F-code level should not be chosen: one could identify dot-products etc. in F-code programs and then implement them using the correct arithmetic unit configuration in a medium-grain manner. The i860 is more dynamic than that. It is expected that the targetter should extract a congruent pipelined schedule for the floating point units by extraction and inference from the T-code representation. This is one justification for retaining the explicit notion of loops in T-code since arithmetic pipelining only occurs within loops. The i860 architecture is particularly fine-grain: the configuration of the arithmetic units may be changed each clock cycle: configuration, operation, and register operands are supplied to the arithmetic units in one floating point instruction.

Floating point instructions advance the floating point pipeline one element at a time, receiving a result a minimum of three instructions (advancements) later. There are separate add and multiply pipelines which may be chained; the configuration is changeable for each advancement. It is a particularly dynamic architecture (one might say that it is not general purpose because it is very difficult to use). Arithmetic pipelining is a large area of research for the future, perhaps meriting a thesis in its own right. A great deal of rewrite operations need to be done in order to transform T-code into i860 instructions with arithmetic pipelining. These will not be as straightforward as the \( \sigma \) and \( \rho \) rewrites.

6.4.3 Superscalar scheduling

One feature of the i860 is the ability to schedule integer and floating point instructions concurrently. Superscalar scheduling cannot go ahead before arithmetic pipelining; but, of course, it can be an independent algorithm, and it can therefore be tested independently. One imagines the implementation proceeding by creating two graphs, each with a spine: a spine of integer (and control) instructions and a spine of floating point operations. Synchronization between the two is at the boundaries of loop bodies, at branch instructions, and secondly at instructions which
6.4. ITERATIVE SCHEDULING AND THE CURRENT IMPLEMENTATION

transfer between the integer and floating point register files. These are the only necessary syn­
chronizations (inter-dependencies) between the two spines. Execution scheduling then occurs on
the integer instructions and floating point instructions for superscalar execution. Floating point
instructions must already have been pipelined to fit the arithmetic units; execution scheduling
cannot re-order the floating point instructions, therefore, but floating point instructions will not
produce execution pipeline bubbles.

No-ops are used to fill gaps where superscalar operation is not warranted or to avoid execution
pipeline bubbles. The i860 has two modes of operation: superscalar mode need not always be
active. There must be some metric to judge when superscalar operation is effective. When
superscalar execution is used, the code is more constituted of no-ops, increasing the requirement,
and throughput requirement, of program memory without necessarily producing a corresponding
increase in computational throughput. The metric is thus easy to choose: merely a comparison
between the number of empty execution slots and the combined integer and floating point
computational throughput, multiplied by some factor.

This is a perfectly adequate model for superscalar scheduling which may be extrapolated for
VLIW architecture, or small numbers of closely coupled processors, synchronous or otherwise.
On asynchronous processors, no-ops are not introduced, of course. Independent spines are
composed in this case with communications foremost. The static choice of the compiler is how
many processors to use to evaluate the computation (or how much computation to assign to a
processor).

6.4.4 Memory pipelining and the cache

The next facet of the i860 which needs to be considered is memory pipelining. There is a three
stage pipeline from data memory to the floating point register file (on the load side). There is
also a direct floating point load. The difference between these types of load is that pipelined
memory load instructions bypass the cache.

There are a number of ways of viewing the cache of the i860: it may be viewed as a number
of vector registers. Thus non-pipelined floating point load and store instructions may be used
solely as references to vector registers. Pipelined load instructions may be used to reference
vectors which cannot be contained in the cache, or for more irregular memory references. The
use of the memory pipeline hides memory latency further, memory accesses which occur via the
cache are slower still in the event of a cache miss.

Obviously it is pointless to hold data in a cache slot which will not be used more than once.
Most of the data in F-code programs are single assignment, which means that data is written
once and subsequently read once. A lazy environment, which is the only type of environment
which does active computation ( apart from creating, destroying, etc. ) on a data parallel
object, reads from a number of single assignment variables and writes the result to one other.
The sequential ordering of lazy environments is a useful guide to cache allocation since caches
are particularly amenable, especially small caches, to short range allocation. Avoidance of cache thrashing is an essential part of sequentialization. A simple hierarchical basis for allocation sections of the cache might be: short-range single assignment variable allocation; ... ;long-range single assignment variables; finally data-objects made by the \texttt{create} function of F-code. Beneficial use of the cache will be maximized by splitting loops into convenient stretches like strip-mining.

One important conclusion of all of this is that the higher-level semantics of F-code provide information for cache allocation, while T-code does not do so easily because it lacks the descriptive-simplicity of F-code — necessarily because it represents the assembly-language implementation of an F-code program. T-code must be extended, and T-code graphs provided with information for cache allocation from the front end of the F-code compiler. The fact that this information may be available in F-code programs is due to their descriptive simplicity and architecture-neutrality a result of inference procedures which may be applied to F-code to infer data lifetimes, which cannot be applied to T-code graphs. Finally, one knows because of what an F-code operation \textit{does} that it may, or may not, make efficient use of the cache depending on its context. This is not so for T-code because it is more low-level. This requires further work.

6.4.5 Register allocation and register coalescing

There are a limited number of registers which means that there will be cases when there are not enough, when over-spill occurs. When over-spill occurs, the instruction stream needs to be modified to include memory loads and stores (the over-spilled register is emulated by cache or main memory.) This means that over-spill modifies the use of the cache, or memory pipelining. It also affects execution pipelining and arithmetic pipelining since extra load/store instructions are introduced to emulate registers. This in turn affects superscalar execution of F-code. Less importantly, around function calls, the active part of the register file must be saved and restored, depending on the chosen calling convention. This means that the instruction stream depends on active registers. This is an important source of optimization in most programs, most important in programs with a high call-ratio.

Register coalescing is used to reduce the number of unnecessary move instructions. Calls to functions have to conform to some calling convention: registers are used to pass operands and to receive results from a function. The T-code graph includes \texttt{coalesce} primitives which trace a particular register number through the data-dependency component of a graph for register allocation, thereby making the operands and results automatically in correct registers (coalescing them).

6.4.6 An iterative approach to low-level scheduling

The discussion has tacitly suggested that execution pipelining, arithmetic pipelining, superscalar scheduling, memory pipelining, and register allocation can be performed consecutively, in some
order. This is not the case. Only an iterative (search) algorithm can produce a truly congruent implementation.

Figure 6.7 shows a rough system for the congruent low-level scheduling of assembly language on an i860. Two throttles on the system are the number of registers and the extent of loop unrolling. The iteration proceeds by first assuming the number of registers available to be the number required, then permits over-spill by progressively limiting the number of registers to those actually available. Other guards for completion of the iteration are finding the most congruent level of loop-unrolling, finding the compound, most efficient use of the arithmetic and memory pipes, and the cache.

Loop unrolling is most effective when loop bodies are small and include use of arithmetic pipelines. Loop unrolling affects the allocation of registers, requiring more to be associated with arithmetic results. Arithmetic pipelining would be very sensitive to the stream of floating point instructions it was supplied with.

All together this is by no means a trivial system. The current implementation is being done in C. This language is not the most apt language in which to implement a system like this. A future implementation will best be attempted in a functional language, or Prolog. This is discussed further in chapter 7.
6.4.7 The initial implementation of the targetter

The previous section delved into future work. The initial implementation is not quite so brave, but does produce i860 assembler. The iterative system above is not implemented. The targetter at this stage is quite straightforward: $\rho$ and $\sigma$ rewrites produce i860 with a spine, registers are allocated, and it is output. This is prerequisite to a final implementation since it aims to test all previous stages of the compiler, and it does so.

The goal of the current compiler is not to produce benchmark-comparable code. It is rather to show how an architecture-neutral platform may be compiled by a series of inference procedures. These have been shown, and most implemented, for an i860. Code quality is on par to some commercial compilers for C.

Real code quality depends on the low-level scheduling of the compiler. This is an area of future work, but one eminently made possible using the iterative approach as guidelined above. The next chapter discusses the major areas of future work, and the overall conclusions about architecture-neutrality, and implementing an architecture-neutral software platform.
Chapter 7

Conclusions, contributions, and future work

7.1 Conclusions

The need for architecture neutrality

A problem which has yet to be implemented on a real machine is not designed with a language, or a machine in mind. It is architecture neutral.

When it comes to implementation, languages very often do not hide the machine at all well from the programmer. The programmer is forced to write code applicable to a particular machine. This is the case for languages which are architecture-specific: for example languages with explicit message passing constructs for distributed machines. This is also the case for languages which require vectorizing compilers: vectorizers are tailored to particular machines and if the programmer wants efficiency s/he may only write code that the vectoriser can vectorize. This means automatically that the program is not portable. Obviously, portability is very important.

It is most often not possible for a programmer to express what parallelism of an application is known; and when it is not known or cannot be expressed it is most often impossible for compilers to infer it. Architecture neutrality implies congruence between a language and a machine, and congruence between the language and the programming problem. This means that a language can best specify the problem and programs written in the language possess portable efficiency.

Portable Software Platforms

It was shown that there will always be any number of high level languages and potentially therefore, a need for any number of compilers for a single machine. This thesis was interested
mostly in high level languages for computational use. What underlies all of these languages is
the expression or compiler-extractability of data-parallelism. A PSP is a generalization of high-
level languages which is used as an intermediate level for compilers. Since it is an intermediate
level a PSP should be explicitly data-parallel (or the parallelism must be inferrable): it should
be able to represent known parallelism.

Not only does a PSP support a number of high-level languages, it also supports all general
purpose parallel machines. The introduction of this thesis showed that a PSP must be
architecture neutral because it supports more than one architecture independent programming
language.

In an architecture neutral way, a PSP generalizes compilation techniques for parallelizing
compilers; collects optimization techniques; provides a means of automatic portability and ef-
ficiency; and provides a fast turn-around for compilers for new high-level languages because
the high-level compilers to a PSP are generally straightforward: this is less so for high-level
languages which require vectorization (this thesis propounds the thought that vectorization of
any but simple programs is impossible anyway). A PSP therefore decouples a large number of
high-level languages from a large number of machines. It can be thought of as a compilation
toolkit; and since the representation is fixed, it can be thought of by compilers for high-level
languages as a 'black box': any compiler which targets to a PSP does not need to know any of
the internals of the PSP, and 'knows' that the PSP will produce code congruent to any general
purpose machine.

F-code

F-code, first described in chapter 2, is a formalism for data-parallelism which possesses algebraic
properties: reflexive transformations (F-code to F-code) can be applied to F-code programs for
optimization. F-code is a PSP which supports imperative languages and hence it needs to
incorporate the imperative constructs and data-types inherent in high-level languages. It is not
therefore a purely functional language since it includes assignment. Importantly however, F-code
provides a scoping mechanism (by means of the F-functions create and so on) which means
that purely functional sections, without side-effects, can be implemented lazily, spatially (across
data-parallel operations.) The generalization of data-parallelism is effected by assuming that all
operations can operate on rectangular data-parallel operands with any dimensionality and any
shape. Shape coercions, when shapes of operands of operations mismatch, are automatically
inserted, and F-code also includes explicit rank coercion operations, and operand orientation —
and together these all mean that F-code is a very general formalism for expressing data-parallel
computation.

Operations are intrinsically data-parallel and do not enforce ordering between elemental op-
erations. This lack of ordering makes spatial lazy evaluation possible. Ordering is less restrictive
still: a lazy data-parallel F-code section can be viewed as a composition of eager scalar F-code
sections which are independent (this is how lazy evaluation is implemented in this thesis on a sequential machine.) These scalar programs can be thought to be like node programs for parallelized execution of F-code, which may be utilized in different ways for different machines: synchronously for SIMD machines; distributely for MIMD machines; or interlacedly, in the manner of loop-unrolling, for VLIW machines, etc.

This unrestrictiveness of ordering, brought about to a large extent by the declarative semantics of F-code, provides a large degree of implementational freedom. This is the basis of architecture-neutrality in F-code. An F-code representation may represent many apparently different high-level programs in a common, most descriptively simple way. Congruent implementations on a very wide variety machines may be inferred from the representation: Data-parallelism may be extracted from the implementation for vector machines, for pipelining, for data-partitioning across distributed machines, etc. A single F-code program may represent any number of apparently different high-level language programs for congruent implementation on any number of general purpose machines. It is an abstraction for both.

An F-code program uses a tree representation, providing the clarity of the scoping mechanism. The representation also therefore reveals functional concurrency: operands of operations may be executed in any order in respect to each other, or arbitrarily interlaced. Functional concurrency is also provided by the view that data-parallel F-code programs are compositions of eager scalar F-code programs which can be scheduled (similarly to loop parallelization) independently or interlaced to overcome latency or to reduce pipeline bubbles, etc.

Static inference

An F-code program is an abstract definition of a problem, with declarative semantics. The process of compiling it to real machines is by a process of inference. When implementing F-code, there are a number of areas of inference: (1) the representation omits all but necessary type, rank, sort and shape information: the compiler must statically infer all of these things (chapter 3); (2) statically inferring the shapes of operations makes lazy evaluation possible: the compiler statically infers which parts of operands are to be evaluated by inferring the shapes of operations and constructing shape expressions (chapter 3); (3) statically inferring an abstract machine graph which implements this program (chapter 5); (4) inferring from this graph a congruent assembly-level language implementation for a real machine (chapter 6.) The first three are architecture-independent parts of the compilation. The last, the targetting inference, is the only part of the compilation procedure which requires any information about the machine.

The front end

The front end of the compiler (chapter 3) infers the type, rank and sort of every node in the F-code tree. It also inserts type coercion nodes into the tree where necessary. The primary result of this is of course that F-code can correctly omit type information from the external
representation and that the compiler can infer it. Type coercion operations are not part of the external definition of F-code: an F-code compiler can create new types of F-instruction for its own compile-time purposes.

The next task of the front end is to infer the shapes of F-expressions. It does so by creating an F-code extent expression for every dimension of every operation in the primary F-expression (which will be called the 'primary' from this point on.) Each extent expression when executed produces a scalar result. Creating so many extent expressions is not as thriftless as it may seem because they are mainly composed of common subexpressions: the particular extent of a node in the primary depends only on the extents of that node's daughters combined in some way. In cases where the extents are known to be constants at compile time, these extent expressions are folded to produce just numbers. If the extents are not known to be constants at compile time, they include references to run-time variables which are set up by executing the primary tree. The use of run-time variables in this way means that there is no need to keep a dictionary of extents for objects. All that this stage of the compilation procedure has done is produce F-expressions which evaluate the extents of nodes in the primary F-expressions.

Selective (Lazy) evaluation

It was shown in chapters 4 and 5 how the extent expressions inferred by the front end can be used to execute F-code lazily. Even when extent expressions are folded to constants, the original connectivity of the expression is maintained. The F-expressions can then be viewed as relationships between indices. Chapter 4 showed how indices are labelled by a straightforward recursion through extent expressions. Chapter 5 then went on to show how geometric operations are executed by scalar index operations: for example, the F-instruction sect is executed by a scalar assignment which sets an index specified to a particular value.

Since F-code has certain fixed shape coercion rules, the extent expressions evaluate the extents of the area of a computation which has to be executed to affect the result. Therefore, in the sequential implementation, loops are generated to scan the data-parallel operation only across these extents. Geometric operations, implemented by scalar index manipulations, mean that relevant parts of operands are evaluated according to the main loop indices: the result of an F-expression is a rectangular object of any dimensionality which is the geometric composition/computation of the expressions operands. This implementation thereby performs the most selective evaluation; and most efficient implementation in the sequential implementation.

For some parallel machines, it may be that using the most selective evaluation does not provide the most efficient implementation. It may be the case that some redundant computation may make the implementation more efficient: for example performing a long vector operation with some redundancy, rather than a number of shorter ones which would incur so many pipeline setup times. The implementation here (chapters 5 and 6) is for a scalar machine, however, where efficiency (congruence) is directly related to selectivity. Importantly, the most efficient
7.1. CONCLUSIONS

implementation of selective evaluation is architecture-sensitive. Selective evaluation on other than sequential machines is a future areas of research

Sequentialization

Chapter 4 showed the coarse scheduling (sequentialization) of F-code programs. This was achieved by describing a process which inferred the position of lazy environments and 'bulk-synchronizations/loops' in F-code trees. An F-code tree is shown to be treated as a number of lazy environments which are enclosed by eager F-functions like create. The treatment is equally applicable to all kinds of machines: distributed, vector, etc.

The chapter shows the necessary sequential order of executing F-code trees. Without rewriting the tree, it also executes the comma function lazily. comma is the F-code function which deals with adding side effects to ordinary lazy, data-parallel expressions. The tree traversal algorithm which produces a sequential ordering which was shown is identical for all types of machine: however the interpretation is a little different. In the scalar sequential implementation (and for SIMD machines), the coarse scheduling algorithm is one level of recursion of the code generation algorithm; this conceptually calls other recursions which produce congruent fine grain scheduling. In the real implementation given in this thesis which targets F-code first via an abstract graph, fine grain scheduling issues are left to the targetter. In the distributed implementation, the coarse scheduling suggests positions of bulk-synchronizations.

Choice of a graph representation

Chapter 5 introduced a graph representation of data-parallel programs which includes all of the functionality of F-code called T-code. In this graph representation are control- and data-dependencies; it is a graph, unlike F-code which is a tree, because it includes branch instructions. It is a graph such, however, that the graph can be traversed from the root without encountering cycles (since some nodes can only be traversed in one direction.) There are other ways of traversing the graph such that only the control arcs are traversed, or only the data dependency arcs are traversed. A traversal of the control arcs is used for scheduling; a traversal of data dependency arcs is used for register allocation, etc. by the targetter: the targetter traces along the dependency tree from the root instruction which writes to a register to all readers, and also to other (exclusive) writers (which are roots or leaves depending on viewpoint.) Control and data dependencies are assimilated in a single graph structure (using the eval function of T-code.) Control dependencies are directed arcs; data dependencies are bi-directional arcs. The fact that data dependency arcs are bi-directional aids register allocation and register coalescing (avoiding redundant move instructions by coalescing the result registers and parameter registers of a function to conform with the i860 calling standard for C-functions and libraries.)

The graph is chosen to subsume the functionality of all RISC processors. It does so so that implementations for other processors apart from the i860 can be created with as little effort as
possible — economy is the art of commercial programming. The F-code compiler is modularized into two programs: the first transforms an F-code tree into a T-code graph; the second program — the targetter — uses only this T-code graph representation and infers from it a congruent implementation on a single RISC processor.

Data arcs of a T-code graph carry scalars of the types of F-code: char..complex, or pointers to non-scalar data. The graph is load-store oriented: it assumes that data is passed between nodes by registers, and that there are explicit load and store instructions. This is the case for RISC processors since they are not microcoded: they do not generally support complex memory-addressing modes. The introduction to chapter 4 discussed the fact that the sequential implementation of the compiler is subordinate to a compiler for a distributed machine. This upholds the fact that arcs in the graph only hold scalar data.

In the past, for example for the VSA, the intermediate level of compilation for software platforms was to a stack based machine. Targetting VSA to a RISC processor would be by transforming the stack based representation to machine instructions, or by emulation. T-code is applicable to stack based machines and RISC processors (less so CISC processors), and maintains sufficient loop information, possibly, for vector execution. The applicability of T-code to machines with vector registers is a future area of research. T-code may need to be extended, to handle these kind of machines.

Graph generation

Chapter 5 showed that the creation of the T-code graph from an F-code program is relatively easy. Most of the implicit variables required to implement an F-code program are single-assignment variables (written to once, read once.) The exception to this are for example: variables which are used to implement F-code's reduce function; and variables for the result of choice which needs the generation of two assignments which are exclusive (one elemental assignment for the true case, and one for the false case, only one of which will be executed for a particular element of the result.)

The T-code graph retains the explicit notion of loops: a loop has an index which increments from zero to some 'extent', along with that a loop increments any number of other 'strides' starting from zero in specified steps. Identifying such strides before producing the T-code tree means that unnecessary, later, possibly inferior, optimization is avoided. The strides are used for example to add an array of integers to an array of complex numbers: one stride increments by the size of an integer, another increments by the size of a complex. This avoids unnecessary multiplications and therefore is very important for efficiency. This approach is far better than a C compiler which expands a reference to an element of a multi-dimensional object with multiplications and additions to find the address and then perhaps attempts to optimize it out.
Generating the functionality of F-code is quite straightforward: T-code includes all of the intrinsic mathematical functions of F-code as is. T-code, however, is a purely scalar implementation of F-code with explicit loops to scan data-parallel operations. Geometric operations are implemented, as was shown, using indices. The extents of the loops are the shape-expressions which were inferred by the front end of the compiler. The shape-expressions also provide the index information necessary to implement F-code lazily.

Targetting

The nodes in a T-code graph do not necessarily correspond exactly to assembler instructions: most do like integer add, floating point add, etc.; others like cosine, log, might not. The T-code graph is rewritten to correspond to assembler instructions by the targetter (chapter 6.) This is the first architecture-specific part of the compilation process. The abstract instructions of the T-code graph are expanded into real i860 instructions. In some cases, instructions of the T-code graph are redundant: for example, loading a byte from memory automatically sign extends it into a word in a register. The T-code instruction which exists to perform this is therefore not needed, and the targetter removes it. Other T-code instructions require an expansion into more than one i860 instruction: for example the abstract instruction which transforms integers to floating point. Some T-code instructions require an expansion into i860 code which calls a library function: for example the cosine and sine functions, etc. There is a trade-off made by the targetter about what should be expanded 'in-line' and what calls should be made. Expanding 'in-line' has the additional benefit that register allocation and fine-grain scheduling can be applied to avoid pipeline bubbles (these cannot be guaranteed of a library call); also of course, saving and restoring allocated registers is not required of 'in-line' functions; detrimentally, however, expanding 'in-line' clutches more memory. Similarly, a T-code loop can be unrolled by replicating the loop block, to avoid pipeline bubbles — with an identical detrimental effect.

Some data-types of T-code may not be supported by the target processor: particularly complex. In such a case, the targetter would rewrite a graph such that complex operations become parallel operations on two separate floating point components. This rewrite also effects further laziness (it introduces further functional concurrency): if only one component of a complex number, say the 'real' part is required to affect the result of a computation, only that component will be evaluated. This is implemented for the i860. Targetting operations like this are best handled by graph rewrites.

This chapter goes on to infer addressing modes for the i860 instructions from the graph representation by tracing data-dependencies; applies a fine-grain scheduler which transforms the graph into a graph with a spine; and performs register allocation. i860 instructions are then the result of the compiler. The current implementation of F-code to an i860 does not make use of arithmetic pipelining or memory pipelining. The scheduler is therefore quite simplistic: more will be said about this in the section on software engineering.
The important conclusion of this chapter is that it is possible to keep architectural concerns towards the end of the compilation process and to still produce a congruent implementation. This allows the machine-specific targetter to be a separate program to the front of the compiler.

Software engineering

This section does not further the virtues of systems analysis, but rather demonstrates that a semi-formal software engineering approach was taken to implement the F-code compiler.

It is not possible to guarantee that any substantial piece of software is flawless, especially something as large as an operating system or a compiler. Compilers may be used to compile code for a great number of systems. Most of these are mundane where flawlessness is only desirable, such as some egghead’s galactic-interaction simulation, where errors have no danger to life or limb. For other environments flawlessness is vital, such as the control software in a fly-by-wire aircraft (which is usually hand-coded.) Software houses would normally never permit legal responsibility for corporeal injury caused by faulty software.

The flawlessness of software is always the goal of an implementor. First of all, the software must have a watertight functional specification. This is available for F-code compilers because F-code is defined using a rigorous algebraic notation. While this notation does not enforce a particular implementation — F-code is architecture-neutral — it provides a well-defined and complete functional specification.

In order to attain a high degree of flawlessness, the software must be tested thoroughly, and the best way of doing so is incrementally — add a function and test it. This is the most fitting strategy for coding a compiler, since as is shown in figure 7.1, compilers are generally ‘wide’ and fairly ‘shallow’: as ‘wide’ as the number of intrinsical functions of the language, and as ‘deep’ as the number of stages of the compilation algorithm. The ‘bottom-up’ method of programming instituted here has the additional benefit that debugging is more straightforward since errors are generally first-order effects — one can also say this of programs written in functional programming languages. Since most of the components of the F-code compiler are independent, classically arranged in compiler stages, and calling on very few shared components (such as the symbol table), a formal systems analysis is not necessary.

The ‘wideness’ of a program does not generally affect the complexity of coding it, even if components in a particular stage of the compiler are shared; increased ‘wideness’ merely adds coding time. Increasing the ‘wideness’ does not introduce flaws into the compiler, except by ordinary, inevitable, simple programming mishaps and mental aberrations: using the wrong variable, calling the wrong function, etc. This is why the psychology of lazy, awake, programmers, who re-use software is the more successful one than that of eager, hasty programmers, who inevitably make mistakes. The perfect programmer is perhaps a machine, which is one reason why compiler generators should be advocated as an attractive area of research for the future. Owners look like their dogs; some programmers think like their machines.
The ‘depth’ of a program introduces debugging difficulty: it is more of a gauge of the coding difficulty. In the case of a compiler, for every intrinsical function of F-code there are components to parse it, to perform type inference on it, to infer the shape of its result, and finally to generate intermediate code for it (T-code.) The coding was such that each stage of the compiler was implemented fully before moving on to the next — and each stage of the compiler produces a useful output. The process of parsing for example produces, if requested, a pretty-printed output. The type-inference stage produces the same output with additional notation for inferred types, and shows explicitly positions of type coercions. Similarly, the shape inference stage may produce all of this with addition shape information — the program in this case will have been rewritten in order for the shape inference to work at run-time and will not be identical to the source program. This makes verification more difficult. Lastly, code generation produces a T-code graph. This stage is very difficult to verify because of the amount of output produced, plus also the lack of a suitable presentation medium (a graphical interface would naturally be most suited for verification.) By verification, here, one means executing test runs that match the programmer’s intuition of what might go wrong in his/her program. It is not possible, of course, to test all cases of a program; programming experience plays its part: it is rather like playing chess against a perfect opponent: well-versed in what can go wrong, looking ahead to how a function may be re-used later in an implementation, one loses by the mistakes one introduces. Programming is a game in which the perfect outcome is a draw.

Along the depth ‘axis’ of a compiler, different programming styles, and strategies are relevant. One would like to assume that a single programming language is suitable for all of the compiler stages; one would also like to assume that this language can be compiled. For the implementation of F-code, due to the different natures of the stages of the compiler, the front-end is perfectly adequately implemented in C, although towards the intermediate code generation stage, verification is difficult to achieve. C is not particularly well suited to graph manipula-
tion, though C can be used to program anything. The F-code compiler is modularized: the
targetting stage is more adequately implemented in a language like Prolog or Miranda because
back-tracking (and the similar ability of functional programming languages) can play a part in
finding the most congruent low-level execution schedule of an assembler program. Writing a C
program that does the equivalent of back-tracking is not trivial; and without substantial effort,
compilers written in imperative languages will be inferior in code quality because of the heavily
pointed way in which they are executed. This is not least the case because low level strategies
of compilers have internecine effects on efficiency, and congruent low-level code generation is, or
should be, an iterative procedure (perhaps even carrying this as far as run-time, and introducing
run-time scheduling): register allocation, scheduling (inc. loop-unrolling, etc.), memory pipelin-
ing, arithmetic pipelining, avoiding cache-thrashing data distribution, etc. are not independent,
and therefore they cannot be done one after the other, in a compiler. This was discussed in
chapter 6.

The coding experience gained in writing the F-code compiler is applicable to any future
implementation and is this: while there is no difficulty in coding the front-end in C, C is the
wrong language in which to implement a targetter. The experience is one of disillusionment
with simple imperative styles of programming for certain applications. Therefore I have become
more of an exponent of applicative languages than ever previously; there is certainly a need
for mixed implementations using imperative and more applicative styles of programming, which
subjectively augurs well for the aims of Portable Software Platforms (more generally extended to
include support for functional programming languages — unlike F-code); and therefore augurs
well for my own thesis.

I have a high degree of confidence in the ability of the front end of the F-code compiler to
produce T-code. This is because F-code was first compiled to scalar C to prove that it can be
compiled (and is portable), and to verify all previous stages of the compiler. The F-code to
scalar C compiler is little different from the F-code to T-code compiler. However, the targetter
has been very difficult to debug because of its complexity. [Maybe I intended to do too much.]
There is no clear method of verification: it is a program whose internal interdependence is such
that either it all works or it all does not: it is the worst kind of ‘depth’ problem. English
puddeny aside, this is a result of the style of programming language used, and not because of a
slack implementation methodology. 

Experto crede.

To summarize, the targetter should be implemented in Prolog or a functional programming
language because (1) they are more appropriate languages in which to manipulate graphs: the
targetting mainly consists of graph-rewriting; (2) to achieve a congruent implementation requires
modelling the performance of the system, and using an iterative search to find the best apparent
compile-time performance. This search, however, must ‘somehow’ not be to the detriment of
compilation time: compiler users must have some handle on the level of search. Search is a
feature of Prolog and naturally of applicative languages.
7.2. FUTURE AREAS OF WORK

There may also be a need for run-time scheduling: on distributed machines this might mean migration of data or program, and on shared-memory machines, the adaptation of memory pipelining.

7.2 Future areas of work

Portable Software Platforms in some guise or other will be of keystone importance to computing in the future. F-code will probably seem more conventional, and more based on high-level languages (less architecture-neutral) than PSPs of the future, but it is the first successful example of its kind, and the implementation given in this thesis is the first implementation of its kind.

PSPs permit multi-lingual implementation of software projects by providing a common underlying programming platform. They overcome the natural human inclination for lack of consensus, in programming languages, by not being overly specific (which is what architecture-neutrality is all about) by using denotational semantics. Clearly, this has commercial as well as practical benefits. While programmers of supercomputers will not ultimately continue to use FORTRAN 77, but will change to FORTRAN 90 and future FORTRAN variants ("whatever programming language will be used in the future, it will be called FORTRAN!"), a PSP provides automatic compatibility between old and new languages. This should encourage the timely change for software houses to FORTRAN 90 without squandering old software. A PSP is something obvious and a little refreshing, so much as to stultify much of current, conventional work on compilers: one extracts what is common amongst high-level languages for parallel computers (data-parallelism), denudes it, expressing it in the simplest way (gaining architecture-neutrality), and implements it on any machine by inferring the most congruent implementation.

FORTRAN 90 is overly sophisticated in the way that data-parallelism is expressed; conversely, PSPs take a minimalist approach while making certain that everything that needs to be expressed is expressible: they express 'perfect parallelism': most notably, vectorization is irrelevant.

PSPs should suggest not only new languages for the future, but also new hardware platforms. It is the case that some compiler software houses prefer complex hardware which is difficult to make the best use of, like Intel processors, because if hardware is overly orthogonal they have little basis on which to compete against each other. Similarly, hardware, if particularly orthogonal can not compete in some applications with hardware primed for a particular purpose. Currently, a general purpose machine is always, therefore, somehow inferior. Thus, although a consensus in high-level languages is highly beneficial, consensus among machines currently makes little commercial sense. Therefore, one area of future work is to use PSPs to design new computer architectures which really are general purpose, and can compete with hardware primed for a purpose — academically it makes sense, and inevitably it will eventually do so commercially to a few pre-eminent vendors.

A PSP representation of a computation is as much a representation of a high-level program
as a machine: it is architecture-neutral. In order to demonstrate the architecture-neutrality of F-code, it needs to be implemented on a number of other architectures: it needs to be implemented on a distributed machine first of all, since this is 'antipodean' to the implementation given in this thesis. As shown in chapter 4, the distributed compiler builds on work already done.

The overall implementation strategy also merits attention. F-code is particularly suited to compiler generators. The way of demonstrating the implementation given in this thesis is not so much to describe low-level algorithms but to demonstrate inferences at particular stages of the compiler. These inference methods are amenable for compiler generators.

As regards the current implementation, the following have been identified as future areas of work:

• The most immediate future work is to rewrite the T-code targetter in Prolog or Miranda, or to write a module which implements the back-track-ability of these languages in C. A more iterative approach to code generation is needed to make code more congruent.

• At the moment, the scoping mechanism does not match the F-code semantics to provide for multiple invocations of functions. This requires the implementation of a dictionary — for function parameters, and automatic variables. A related topic is that recursion has yet to be implemented in this F-code compiler. Recursion and multiple invocations of functions modify shape inference, as described in section 3.9.

• Due to the lack of time, particularly frilly pipelining techniques have not been applied at the code generation stage. This is a large area of future work, one which might warrant another thesis. The current implementation produces assembly language. Optimization of this involves the use of an iterative scheduler. I have not been interested in benchmarks but have largely shown that benchmark-comparable code will ultimately be possible.

• The implementation of selective evaluation on other than sequential machines. The current implementation makes use of the most selective evaluation. As discussed previously in this chapter, selective evaluation is often architecture-sensitive. The sequential ordering given in chapter 4 also indicated it is possible to pipeline between adjacent lazy environments, which is very important for an implementation of F-code on a distributed machine.

• The applicability of T-code to machines with vector registers is a future area of research; perhaps there is a need to extend it for types of machine other than just RISC processors.

More generally, for F-code itself,

• F-code is not fully architecture-neutral. It is not suitable for representing irregular problems. Increased congruence might also imply run-time compilation for irregular problems. F-code needs to be extended to handle irregular problems.
• The use of the algebraic properties of F-code for architecture-dependent optimization has not been fully investigated. A rudimentary set of optimizations were given in chapter 3.

• It might seem reasonable to say that for a distributed implementation of F-code, data-partitioning directives might need to be added to the external F-code representation. This is incompatible with architecture-neutrality, however, and it must be remembered that F-code compilers (all compilers for architecture-neutral languages) should be interactive if data-partitioning cannot be made fully automatic. The argument for this was given in chapter 1.

7.3 Contributions

Very finally, it is useful to conclude with a statement of what, in my unassuming opinion, are the contributions of this thesis. The cornerstone of this thesis is the PSP, F-code. F-code was defined by Shafarenko, Muchnick, and Bolychevsky, after extensive work in data-parallel programming languages, and compilers, done in Russia; it is based on substantial experience of users’ requirements for data-parallel primitives, and contains virtually all of the primitives shown later to be required by the programming community in HPF correspondence and documentation. Several F-functions will be added shortly to the definition — which do not change this work in any fundamental way. The basis of this thesis — F-code — is thus sound, presumably complete (which is a rarity), and thoroughly pragmatic.

The most abstract contribution of this thesis was to show that F-code, hence this kind of PSP, are compilable. This was achieved by implementing an F-code to scalar C compiler. Since C is portable (if not efficiently on all machines) this shows, in a small way, that F-code is portable. However, since at no stage during the compilation is parallelism discarded, and it is always explicit, and manipulable, F-code can obviously be implemented in any number of ways and on all general purpose machines. A particular implementation has been shown from the architecture-independent way that the front-end is implemented to the architecture-specific way that targeting takes place. The point of showing the complete implementation is that, demonstrably, architecture-specific aspects of the compiler can be deferred as late into the compilation process as the targeting stage of T-code was. While T-code itself is not particularly novel or interesting (there is no reason for it to be), it demonstrates the obvious idea of retaining parallelism and known dependencies until the very end of the compilation process where it can be made use of to utilize architectural features of a machine — such as arithmetic pipelines and VLIW-style execution units.

The implementation of F-code is lazy (selective). The treatment of both left- and right-hand sides of assignments can be selective and include geometric — and in the case of the right-hand side also arithmetic — functions, and this poses no extra difficulty to the compiler [writer]. This is an extra and important source of parallelism which ought to be integrated into conventional
languages since, after all, it poses no additional difficulty — and is a useful optimization available to programmers. While the compiler implements lazy evaluation, it is still scheduled eagerly.

The other contributions are less principal, and deal with the implementation: It is possible to invent an algorithm for shape inference which infers the shape of every F-code operation in a program: the shapes inferred for a particular program are F-expressions themselves. This inference is possible because F-code is defined rigourously, and shape-coercion is an important aspect of F-code, as it should be of any data-parallel language. This inference also permits dynamically shaped objects (objects whose size is only known at run time.) This does not affect scheduling. It is possible to infer the size of all objects before any manipulation takes place on them. The shape expressions are also known to be means of assigning indices to F-functions, to say which index does what in the implementation. Once these indices are known it is possible to apply loop-invariant removal on F-trees (though this is not addressed in this thesis).

The thesis shows the sequentialization ordering of an F-tree. It shows that it is possible to include a data-parallel functional subset of a language in an imperative setting. It shows how commas which are part of the functional (lazy) subset of F-code are sequentialized out of the functional part of a program, and are evaluated before or after the lazy environment. The lazy environment part of a segment of F-code is thus referentially transparent.

The code generation part of this thesis shows how F-code can be transformed into T-code (the transformation into C is very similar) which is a scalar representation of the code's algorithm. The thesis goes on to show that T-code can be targetted to any RISC machine, by moulding it onto the architecture. This requires a congruence model of the architecture, and an iterative procedure to find the best congruent implementation of a program; this is a vast area of future work.

To finish this thesis, it is useful to compare the compilation of FORTRAN 90, and F-code. FORTRAN 90 does not include the shape coercion rules of F-code, which are very important for the method I have given of compiling F-code. FORTRAN 90 is not compiled lazily (functions cannot be mingled together in lazy sections, and their elemental computations done independently without the programmer explicitly programming this.) Identifying side-effects is as complicated in FORTRAN 90 as in any ordinary high-level imperative language. In F-code it is different: side-effects can be identified easily, and sequentialized such that functional sections of F-code be readily extracted. It would be incredibly messy to write a FORTRAN 90 compiler which contains the inherent optimizations of an F-code one.

The last contribution of this thesis is, perhaps, a sad little brown patch of ground where stood the pulp you're clutching. But let's not fret about it.
Appendix A

Definition of F-code

This appendix includes a formal definition of F-code. This section is a direct excerpt of [BMS92], included to give a formal description of F-code. The data-parallel algebra is used in chapter 3 to describe data-parallel optimizations. This appendix is also used as a reference point for the implementation.

It details some specific features of F-code, and is used in order to demonstrate the way F-code is defined as well as to show a few interesting solutions which have been found to the problem of accurate representation of data-parallelism in an architecture-neutral manner.

A.1 Objects

The F-functions receive as operands, and return as results, primitive objects, which are homogeneous, scalar or data-concurrent nonscalar, aggregate of scalar elements having the following characteristics:

1. Type $t \in \{\text{logical, character, integer, real, complex}\}$. The types form an ascending hierarchy in the same order as they are listed above. For example, a character type is junior to the real type and senior to the logical type.

2. Sort $s \in \{\text{value, name, target}\}$. If an object is of sort value, it contains data of type $t$; if it is a name, every element of it contains a reference to a value element of type $t$; if it is a target, every element of it contains either a reference to a value element of type $t' \geq t$ or a special dummy reference. Targets are used for element-wise movement of values in the course of assignment. A name can be used the same way as a target, but also admits data-parallel dereferencing, yielding a value of type $t$.

3. Shape $d = \{d_i\}$ It is a vector of the extents of each of the object's dimensions $d_i$, each of which is a nonnegative integer number. The length of $d$ is equal to the object's rank. In particular, if the object is scalar, vector $d$ has zero length.

4. Contents $c$, which form an array of shape $d$.

---

1 In this thesis we assume that an index of any kind starts with zero.
The F-program can manipulate heterogeneous aggregates of data as well, such as Pascal records or C structures. Characteristics of such structured objects are as follows:

1. Template (metatype) $T$, which is a tree whose leaves are types; all the nodes of this tree except the root one are labelled with shape vectors. Thus, at each level of hierarchy a template determines a sequence of fields. The number of the fields is equal to the number of successors of the node, while the types (or templates) and shape vectors of those fields are defined by the corresponding leaves (or subtrees) and their labels.

2. Shape $d$.

3. Contents $c$, which form an array of shape $d$ built up from structures of template $T$.

Structured objects can only consist of values. However, while accessing their homogeneous fields, primitive objects of any sort may arise. Such an access is effected through pointers, which are primitive objects of integer type. Pointers are also used for references to subroutines.

A.2 Representation of the F-program

The F-program is a single expression to be evaluated. It is represented in a LISP-like list form. Two kinds of atoms are used:

Literals. A literal is a self-defined entity having a regular syntax. Its semantics is unambiguously and statically determined by the syntax derivation. For example, 3.5 is a literal representing real number 3.5 and the literal `mul` denotes multiplication.

Identifiers. The identifier is the only semantically variable entity in the F-language. They are associated with primitive objects, pairs, templates or channels.

While defining syntax of the language, we will use the standard BNF notation with the only extension: ellipsis $\ldots$ stands for any number of repetitions of the preceding item (or the braced group of items).

A.3 Identifiers

The identifier is composed of reasonably large set of characters. It need not start with an alphabetic symbol.

$\langle$identifier$\rangle ::= \langle$symbol$\rangle$...

$\langle$symbol$\rangle ::= \langle$h-digit$\rangle |$

  G | H | I | J | K | L | M | N | O | P |

  Q | R | S | T | U | V | W | X | Y | Z |
A.4. LITERALS

A.4.1 Constants

Apart from traditional constants, the set of F-constants includes machine specific characteristics and pointers to subroutines.

<constant> ::= <logical> | <character> | <string> |
              <integer> | <real> | <complex> | <special> |
              <subroutine>

<logical> ::= true | false

<character> ::= 'ASCII'

<string> ::= "ASCII"

<integer> ::= <decimal> | <octal> | <hexadecimal> | <bit>

<decimal> ::= <number> | <sign><number>

<sign> ::= + | -

<number> ::= <significant> | <significant><d-digit>...

<octal> ::= 0<o-digit>...

<hexadecimal> ::= 0X<h-digit>.. | 0x<h-digit>...

<bit> ::= 0B<mask> | 0b<mask>

<mask> ::= <b-digit>...

<b-digit> ::= 0 | 1

<o-digit> ::= <b-digit> | 2 | 3 | 4 | 5 | 6 | 7
APPENDIX A. DEFINITION OF F-CODE

<digit> ::= <o-digit> | 8 | 9

<significant> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<h-digit> ::= <d-digit> |
A | B | C | D | E | F | a | b | c | d | e | f

<real> ::= <unsigned> | <sign><unsigned>

<unsigned> ::= <number>.<number><exponent> |
<number>.<number> |
<number><exponent>

<exponent> ::= E<decimal> | e<decimal>

<part> ::= <number> | <unsigned>

<complex> ::= <part><sign><part>i | <sign><part><sign><part>i

<special> ::= top | epsilon | max-int | max-char | hole

<subroutine> ::= <EXPR>

Notes:

1. As there is no point in defining the standard ASCII character set, we simply included a non-terminal token <ASCII> denoting an arbitrary ASCII character. Quote " should be doubled if it occurs as an <ASCII> in a string.

2. Non-terminal <EXPR> is defined in section A.6 and generates F-expressions.

These are examples of legal constants and like entities that we shall use further:

1. <logical>: false
2. <character>: 'a' 'b' 'c'
3. <string>: "12abc0#%" "This is string ""X"""
4. <number>: 12 25
5. <decimal>: 12 -25
A.4. LITERALS

6. <octal>: 013 07777

7. <hexadecimal>: 0X12A3F 0x12a3F 0x12a3f

8. <mask>: 1011 0001 000

9. <bit>: 0b100111 0b1111011

10. <real>: 3.56e10 -12.2E-3 -12.2e-3 +25.9

11. <complex>: 1.5+2.7i -3e7+0.1i 2-3i

12. <special>: epsilon

13. <subroutine>: (dyadic add (var value X) (const 1))

A.4.2 Operators

These literals denote all the element-wise operators that are supported by the F-language:

<unary> ::= neg | inv | frac | sqrt | arg | conj | modulus |
         sin | cos | tan | arcsin | arccos | arctan |
         sinh | cosh | tanh | exp | ln |
         even | odd | not | bit-not |
         round | trunc | font

<total> ::= add | mul | max | min |
          and | or | xor | bit-and | bit-or | bit-xor

<binary> ::= <total> | sub | div | idiv | modulo |
           pow | ipow | log | shr | shl | ror | rol |
           gt | ge | lt | le | eq | ne

To avoid any ambiguity, the following table defines the semantics of operators. For the sake of convenience let us denote the operand of a unary operation as \( x \) and the operands of the binary operation as \( x \) and \( y \).
APPENDIX A. DEFINITION OF F-CODE

Literal   Definition
neg       $-z$
ref       $z$
inv       $1/z$
frac      $z - \lfloor z \rfloor$
sqrt      $\sqrt{z}$
arg       $\arg z$
conj      $z^*$
modulus   $|z|$
sin       $\sin z$
cos       $\cos z$
tan       $\tan z$
arcsin    $\arcsin z$
arccos    $\arccos z$
arctan    $\arctan z$
sinh      $\sinh z$
cosh      $\cosh z$
tanh      $\tanh z$
exp       $e^z$
ln        $\ln z$
even      true only if $z$ is even
odd       true only if $z$ is odd
not       $z$
bit-not   bit-wise $z$
round     round $z$
trunc     truncate $z$
font      character with code $z$

Literal   Definition
add       $x + y$
mul       $xy$
max       $\max(x, y)$
min       $\min(x, y)$
and       $x \& y$
or        $x | y$
xor       $x \oplus y$
bit-and   bit-wise $x \& y$
bit-or    bit-wise $x | y$
bit-xor   bit-wise $x \oplus y$
sub       $x - y$
div       $x / y$
idiv      $x / y$, error if $x \pmod{y} \neq 0$
modulo    $x \pmod{y}$
pow       $x^y$, $x$ and $y$ must be real
log       $\log_x y$, $x$ and $y$ must be real
ipow      $x^y$, $x$ is arbitrary, $y$ must be integer
shr       shift $x$ $y$ times right
shl       shift $x$ $y$ times left
xor       rotate $x$ $y$ times right
rol       rotate $x$ $y$ times left
lt        $x < y$
ge        $x \geq y$
le        $x \leq y$
eq       $x \neq y$
ne        $x \neq y$
The following tables define admissible operand types and the result type for the above operators. Every item in the first column specifies a group of operators with the same rule of determining the result type. Letters \( L, H, I, R, C \) and \( U \) stand for an operand of logical, character, integer, real, complex, or "unclear" type, respectively. The corresponding item in the second column indicates the result type versus the unclear operand type in the increasing order. For example ".. I R R" would mean that any operation belonging to the group is undefined for \( U \) being logical or character, and will yield an integer result if the unclear-type operand happens to be integer, and a real result if it happens to be real or complex.

### Arithmetic

<table>
<thead>
<tr>
<th>( U ) add ( U )</th>
<th>( U ) sub ( U )</th>
<th>( U ) neg ( U )</th>
<th>( U ) mul ( U )</th>
<th>( . . I R C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U ) div ( U )</td>
<td>( U ) inv ( U )</td>
<td>( I ) mod ( I )</td>
<td>( . . R C )</td>
<td>( \text{integer} )</td>
</tr>
<tr>
<td>( \text{modulus} U )</td>
<td>( \text{frac} R )</td>
<td>( \text{sqrt} R )</td>
<td>( R ) pow ( R )</td>
<td>( . . ) I R R</td>
</tr>
<tr>
<td>( U ) ipow ( I )</td>
<td>( \text{conj} C )</td>
<td>( \text{arg} C )</td>
<td>( \text{integer} )</td>
<td>( \text{complex} )</td>
</tr>
<tr>
<td>( U ) max ( U )</td>
<td>( U ) min ( U )</td>
<td>( I ) shr ( I )</td>
<td>( I ) shl ( I )</td>
<td>( I ) rol ( I )</td>
</tr>
<tr>
<td>( I ) rol ( I )</td>
<td>( . . ) I R R .</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Basic Functions

<table>
<thead>
<tr>
<th>( \sin R )</th>
<th>( \cos R )</th>
<th>( \tan R )</th>
<th>( \text{real} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{arcsin} R )</td>
<td>( \text{arccos} R )</td>
<td>( \text{arctan} R )</td>
<td>( \text{real} )</td>
</tr>
<tr>
<td>( \sinh R )</td>
<td>( \cosh R )</td>
<td>( \tanh R )</td>
<td>( \text{real} )</td>
</tr>
<tr>
<td>( \ln R )</td>
<td>( R \log R )</td>
<td>( \text{real} )</td>
<td>( \text{real} )</td>
</tr>
<tr>
<td>( \exp U )</td>
<td>( \text{... R C} )</td>
<td>( \text{... R C} )</td>
<td>( \text{... R C} )</td>
</tr>
</tbody>
</table>

### Comparisons

<table>
<thead>
<tr>
<th>( U ) gt ( U )</th>
<th>( U ) ge ( U )</th>
<th>( U ) lt ( U )</th>
<th>( U ) le ( U )</th>
<th>( . . L L . . )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U ) eq ( U )</td>
<td>( U ) ne ( U )</td>
<td>( \text{L L L L L} )</td>
<td>( \text{L L L L L} )</td>
<td>( \text{logical} )</td>
</tr>
<tr>
<td>( \text{even} I )</td>
<td>( \text{odd} I )</td>
<td>( \text{logical} )</td>
<td>( \text{logical} )</td>
<td>( \text{logical} )</td>
</tr>
</tbody>
</table>

### Logic

<table>
<thead>
<tr>
<th>( \text{not} L )</th>
<th>( L ) and ( L )</th>
<th>( L ) or ( L )</th>
<th>( L ) xor ( L )</th>
<th>( \text{logical} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{bit-not} I )</td>
<td>( \text{bit-and} I )</td>
<td>( \text{bit-or} I )</td>
<td>( \text{bit-xor} I )</td>
<td>( \text{integer} )</td>
</tr>
</tbody>
</table>

### Descents

<table>
<thead>
<tr>
<th>( \text{re} C )</th>
<th>( \text{im} C )</th>
<th>( \text{real} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{round} R )</td>
<td>( \text{trunc} R )</td>
<td>( \text{integer} )</td>
</tr>
<tr>
<td>( \text{font} I )</td>
<td>( \text{character} )</td>
<td>( \text{character} )</td>
</tr>
</tbody>
</table>
Note, that the set of admissible types for an operand of an unclear type is always a subrange so that it can be characterized with the junior $t^j$ and senior $t^s$ admissible types.

A.4.3 Keywords

The following literals are used by F-functions as keywords:

$<$type$> ::= \"logical\ | \"character\ | \"integer\ | \"real\ | \"complex$

$<$sort$> ::= value \ | target \ | name

$<$access$> ::= <sort> \ | pointer

$<$property$> ::= <sort> \ | couple

$<$selector$> ::= re \ | im

A.4.4 Function Labels

All the function labels are literals as well.

A.5 Evaluation of the F-program

The evaluation rules are as follows:

1. The first item of the function list is a literal determining the function to be applied to the other items of the list.

2. Subsequent items are literals and identifiers the function needs to specify the activity it is to provide, or the arguments of the function, which are expressions that the F-system evaluates prior to the function invocation, unless otherwise stated in the function definition.

3. Any F-function that assumes its arguments to be evaluated before its invocation does not require any specific order of that evaluation. It ends up by forming operand objects, which we call a priori operands. The F-function deals with a posteriori operands, which are obtained from the a priori ones as a result of optional type coercion. A coercion changes the operand type; a value going up the hierarchy generalising the contents element-wise, a target going down with the contents preserved. A name can not be coerced.

While defining the function semantics, we will specify the a posteriori types of all of the operands. It will be assumed that if the a priori type of an operand happens to be different, then the coercion will normally take place. If the coercion is impossible, eg if it should attempt to lower the type of a value or change the type of name, this will result in an error.
A.6 Functions

In this section we present a list of function definitions. Construct EXPR enumerates all the functions and is introduced to recursively close the F-code syntax.

\[
EXPR ::= \text{CONST} \mid \\
\text{HOLD} \mid \text{TEMPLATE} \mid \text{CREATE} \mid \text{VAR} \mid \text{SELECT} \mid \\
\text{GLOBAL} \mid \text{DISPOSE} \mid \text{LOCAL} \mid \text{MARK} \mid \\
\text{RAMP} \mid \text{MONADIC} \mid \text{DIADIC} \mid \text{CHOICE} \mid \text{REDUCE} \mid \\
\text{TRANS} \mid \text{SECT} \mid \text{SLICE} \mid \\
\text{REPL} \mid \text{PACK} \mid \text{GATHER} \mid \text{DIAG} \mid \text{TRANSFORM} \mid \\
\text{PART} \mid \text{COMP} \mid \text{POL} \mid \\
\text{DISPLACE} \mid \text{DISTANCE} \mid \\
\text{TYPE} \mid \text{SHAPE} \mid \\
\text{ASSIGN} \mid \text{CHANNEL} \mid \text{PUT} \mid \text{GET} \mid \\
\text{SEQ} \mid \text{COMMA} \mid \text{PAR} \mid \text{LOOP} \mid \text{SPAWN} \mid \text{IF} \mid \\
\text{CALL} \mid \text{COERC} 
\]

We chose to define the semantics by providing mathematical formulas which evaluate the characteristics of the result object from those of operands. Here are some auxiliary formulas we are going to use for that purpose. They are to do with some transformations of, and relations between, coordinate or shape vectors we use to define the semantics of the F-language, rather than with any nonscalar objects the F-language manipulates.

1. The contents array of a primitive object is indexed with a multi-index, which is a vector of nonnegative integers. Let us denote the length of \( a \) as \( r(a) \) and introduce a partial precedence of such vectors. We will say that a vector \( a \) precedes a vector \( b \) if \( r(a) = r(b) = r \) and \( a_k < b_k \) for all \( 0 < k < r \). We should denote this as \( a < b \). All the other operations with vectors that we will use below should be interpreted element-wise.

2. For a vector \( a \) of some length and and a mask \( m \) of the same length we introduce a projector as follows:

\[
P : (a, m) \rightarrow b,
\]

where \( b_k = a_{x(k, m)} \), and \( x(k, m) \) is the number of the \( k \)-th unity bit in the mask \( m \). The length of the result vector is equal to the number of unity bits in the mask.

3. Let us introduce an expander:

\[
E : (a, m) \rightarrow b,
\]

so that

\[
b_k = \begin{cases} 
\infty & \text{if } m_k = 0 \\
ax(k, m) & \text{otherwise}
\end{cases}
\]
where $X(k, m)$ is the number of mask bits $m_j = 1$ with $j < k$. Expander $E(a, m)$ is determined only if $m$ has the number of unity bits equal to the length of $a$. The length of $b$ is equal to the number of bits in mask $m$. Thus the following identity takes place:

$$P(E(a, m), m) \equiv a.$$ 

To facilitate references to different function arguments and operands obtained in their evaluation, we will use a special suffix $\cdot x$ with some character $x$. Letters in formulas may use the same character as a superscript. Also the following shorthand notations will use the same character:

$$p^x(k) = P(k, m^x),$$

$$\bar{p}^x(k) = P(k, \overline{m^x}),$$

$$e^x = E(d^x, m^x),$$

where $m^x$ is a mask preceding the argument $EXPR.x$ and $\overline{m^x}$ is the result of inversion of $m^x$:

$$\overline{m^x} = \begin{cases} 0, & m^x_i = 1 \\ 1, & m^x_i = 0 \end{cases}.$$ 

Let us denote as $s^x(k)$ an array containing all the elements of some layer of operand $EXPR.x$ contents. Its shape is $d = p^x(d^x)$ and its elements are determined by the following formula:

$$s^x(k)_l = e^{x} = e^{x}_{\min(E(k,\overline{m^x}),E(l,\overline{m^x}))}, \quad 1 < d.$$ 

A.6.1 const

This function returns an object, whose type, sort, shape and contents are determined by a literal.

CONST ::= ( const <constant> )

Depending on the <constant> the result is as follows:

1. <logical>: A logical scalar value with the contents specified.
2. <character>: A character scalar value containing the character specified between apostrophes.
3. <string>: A one-dimensional character value with the contents and shape determined by the sequence of characters between quotes. All the doubled quotes in this sequence are replaced by single ones.
4. <integer>: An integer scalar value containing the number specified.
5. <real>: A real scalar value containing the number specified.
6. <complex>: A complex scalar value containing the number specified.
7. top: A real scalar value that contains machine-specific constant $\tau$, such that $\tau$ and $\frac{1}{\tau}$ approximate the upper and lower boundaries of the floating-point range.

8. epsilon: A real scalar value that contains machine-specific constant $\epsilon$, being the minimal floating point number for which $1 + \epsilon > 1$ and $1 - \epsilon < 1$.

9. max-int: An integer scalar value containing the maximal integer number supported by implementation.

10. max-char: A character scalar value containing the maximum code character belonging to the character set of the implementation.

11. hole: A complex scalar target containing the dummy reference. Its a priori complex type enables coercion to any a posteriori type.

12. <subroutine>: An integer scalar value containing a pointer to the subroutine defined by the F-expression specified.

A.6.2 hold

This function arranges a scope for a primitive object. It does not assume its first argument to be precomputed.

$$\text{HOLD ::= ( hold <identifier> EXPR.a EXPR.i )}$$

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.

2. The identifier is associated with the EXPR.i operand.

3. The argument EXPR.a is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.

4. The object associated with the identifier is destroyed.

5. The current association for the identifier is destroyed and the saved one (if any) is restored.

6. Operand EXPR.a is returned.

A.6.3 template

This function arranges a scope for a template. It does not assume its first argument to be precomputed.

$$\text{TEMPLATE ::= ( template <identifier> EXPR.a FIELD ... )}$$

$$\text{FIELD ::= MODE EXPR.e ...}$$

$$\text{MODE ::= <type> | <identifier>}$$
The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.

2. The identifier specified after the function label is associated with a template that is built in the following way. The number of successors of the template root is determined by the number of the fields specified. If the type alternative is chosen as mode, then the node in question is a leaf of that type. If the identifier alternative is chosen, then this identifier must be associated with another template; in that case the node is the root of a subtree, which is a copy of that template. In both cases all the operands $\text{EXPR.}_i, 0 \leq i < n$, must be scalar values of a posteriori integer type, and the label of the node is determined as follows:

$$d_i = e^{t_i}, \ 0 \leq i < n.$$  

3. The argument $\text{EXPR.a}$ is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.

4. The current association of the identifier is destroyed and the saved one (if any) is restored.

5. Operand $\text{EXPR.a}$ is returned.

A.6.4 create

This function creates a primitive or a structured variable and arranges a scope for it. It does not assume its first argument to be precomputed.

$$\text{CREATE} ::= ( \text{create} \ <\text{identifier}> \ \text{EXPR.a} \ \text{MODE} \ \text{EXPR.e} \ldots )$$

$$\text{MODE} ::= <\text{type}> | <\text{identifier}>$$

Operands $\text{EXPR.e}_i, 0 \leq i < n$, must be scalar values of a posteriori integer type.

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.

2. Depending on the alternative chosen as a mode, one of the following is done:

   (a) $<\text{type}>$: The primitive value of this type having arbitrary contents is created; its shape is determined as follows:

   $$d_i = e^{t_i}, \ 0 \leq i < n,$$  

   Then this value is enreferenced by creating a name with the same type and shape that refers to the value element-wise. The identifier specified after the function label is associated with a couple that includes the original value and the result of its enreferencing.
(b) `<identifier>`: This identifier must be associated with a template; the structured object of that template having arbitrary contents is created; its shape is determined in exactly the same way as in `type` case. Then the integer scalar value is created being a pointer to this structured object. The identifier specified after the function label is associated with this pointer.

3. The argument `EXPR.a` is computed using the new association for the identifier. Its a posteriori type coincides with the a priori one.

4. The objects created during execution of the step 2 are destroyed.

5. The current association for the identifier specified after the function label is destroyed and the saved one (if any) is restored.

6. Operand `EXPR.a` is returned.

A.6.5 `var`

This function returns an object using some identifier.

\[ VAR ::= ( \text{var} \text{<access>} \text{<identifier>} ) \]

Depending on the keyword the result is computed as follows:

1. The keyword is `value`

   (a) If the identifier is associated with a value, this value is returned.

   (b) If the identifier is associated with a name, the result of dereferencing the name is returned. This result is a value with the same type and shape as the original name. Each element of the newly created value has the same contents as the one referred to by the corresponding element of the original name.

   (c) If the identifier is associated with a couple, the value belonging to this couple is returned.

   (d) Otherwise this results in an error.

2. The keyword is `name`

   (a) If the identifier is associated with a name, this name is returned.

   (b) If the identifier is associated with a couple, the name belonging to this couple is returned.

   (c) Otherwise this results in an error.

3. The keyword is `target`

   (a) If the identifier is associated with a target, this target is returned.
APPENDIX A. DEFINITION OF F-CODE

(b) If the identifier is associated with a name, the target having the same type, shape and contents is returned.

(c) If the identifier is associated with a couple, the target having the same type, shape and contents as the name belonging to this couple is returned.

(d) Otherwise this results in an error.

4. The keyword is pointer

(a) If the identifier is associated with a couple, the integer scalar value being a pointer to the value belonging to this couple is returned.

(b) Otherwise this results in an error.

A.6.6 select

This function performs access to the field of structured object.

\[
\text{SELECT ::= ( SELECT } \langle \text{access} \rangle \langle \text{identifier} \rangle \langle \text{number} \rangle \ \text{EXPR.p )}
\]

The operand must be a posteriori integer value; its elements are treated as pointers to structured objects having template \( T \) associated with the identifier specified. The literal \(<\text{number}>\) defines the ordinal number \( N \) of the field to be selected; The root of \( T \) must have at least \( N \) successors.

The elementary selection with respect to an element of the operand is defined as follows:

1. If the \( N \)-th successor of the root of \( T \) is in its turn a root of some subtree, i.e. the corresponding field is structured, then the integer scalar value is obtained being a pointer to that field (in this case \text{pointer} must be specified as an access).

2. If the \( N \)-th successor is a leaf, i.e. the corresponding field is primitive, then either a copy of this field is obtained (if the \text{access} value is specified) or the \text{target/name} referring to this field element-wise (if the \text{access} specified is \text{target/name, respectively), or the pointer to this field (if the access is \text{pointer}).

Let \( d^F \) be the common shape vector of all elementary selections: the empty vector if a structured field is selected or the label of the corresponding node of \( T \) in the case of a primitive field. The function result is computed as follows:

\[
d = \min \left( E(d^p, m), E(d^F, \tilde{m}) \right),
\]

\[
c_k = F(c^p_{P(k,m)} P(k,*)), k < d,
\]

where

\[
m_i = \begin{cases} 
1, & 0 \leq i < r(d^p) \\
0, & r(d^p) \leq i < r(d^p) + r(d^F)
\end{cases}
\]

and \( F(x) \) is the result of element selection using the pointer \( x \).
A.6.7 global

This function creates a primitive or a structured object in the global heap memory.

GLOBAL ::= ( global MODE EXPR.e ... )

MODE ::= <type> | <identifier>

If an identifier is specified as a mode then it must be associated with a template. Operands EXPR.ei, 0 ≤ i < n, must be scalar values of a posteriori integer type.

The primitive value of the type specified (if type alternative is chosen as a mode) or the structured object of the template specified (if identifier alternative is chosen) having arbitrary contents is created in the global heap memory. The shape of this object is determined as follows:

\[ d_i = e_i, \ 0 \leq i < n, \]

Then the integer scalar value is created being a pointer to newly created object. This pointer is returned as the function result.

A.6.8 dispose

This function disposes the object previously created by GLOBAL function. It always returns integer scalar value 0.

DISPOSE ::= ( dispose EXPR )

The operand must be an a posteriori integer scalar value being a pointer to the object allocated in the global heap memory by GLOBAL function.

A.6.9 local

This function creates a primitive or a structured object in the local heap memory.

LOCAL ::= ( local MODE EXPR.e ... )

MODE ::= <type> | <identifier>

If an identifier is specified as a mode then it must be associated with a template. Operands EXPR.ei, 0 ≤ i < n, must be scalar values of a posteriori integer type.

The primitive or the structured object having arbitrary contents is created according to exactly the same rules as in GLOBAL function. The only difference is that this object is created in the local heap memory.

Then the integer scalar value is created being a pointer to newly created object. This pointer is returned as the function result.
A.6.10 mark

This function marks the levels of locality of the local heap memory. It does not assume its argument to be precomputed.

\[
\text{MARK} ::= ( \text{mark} \ EXPR.a )
\]

The function is executed as follows:

1. The state of the local heap memory is stored.
2. The argument \(EXPR.a\) is computed. Its a posteriori type coincides with the a priori one.
3. All the objects created in the local heap memory while computing the argument are disposed.
4. The operand \(EXPR.a\) is returned.

A.6.11 ramp

This function returns a one-dimensional value containing arithmetic progression, whose parameters are determined by the operands.

\[
\text{RAMP} ::= ( \text{ramp} \ EXPR.b \ EXPR.e \ EXPR.s )
\]

All the operands must be scalar values. Their a posteriori types are determined as follows:

\[
t^b = t^e = t^s = \min (\text{real}, \max (t^b, t^e, t^s, \text{integer})) .
\]

The shape and contents are as follows:

\[
d_0 = \left\lceil \frac{c^e - c^b}{c^s} \right\rceil + 1,
\]

\[
c_k = c^b + k c^s, \quad 0 \leq k < d_0.
\]

A.6.12 monadic

This is a function performing a data parallel monadic operation.

\[
\text{MONADIC} ::= ( \text{monadic} \ <\text{unary}> \ EXPR.a )
\]

The operand must be a value. Its a posteriori type is this:

\[
t^a = \min (t^\mathbb{S}, \max (t^a, t^J)) ,
\]

where \(t^\mathbb{S}\) and \(t^J\) are the senior and junior admissible types of the operator \(<\text{unary}>\), respectively.

The result is computed as follows:

\[
d = d^a ,
\]

\[
c_k = \odot a_k , \quad k < d ,
\]

where \(\odot\) is the operation determined by \(<\text{unary}>\).
A.6. FUNCTIONS

A.6.13 dyadic

This is a function performing a data parallel dyadic operation.

\[
\text{DYADIC} ::= ( \text{dyadic} \text{ <binary> <mask> EXPR.1 <mask> EXPR.r } )
\]

Both operands must be values. Their a posteriori types are as follows:

\[
\hat{t} = \hat{t'} = \min\left(t^s, \max\left(t^l, t^l', t^l\right)\right),
\]

where \(t^s\) and \(t^l\) are the senior and junior admissible types of the operator \(<\text{binary}>\), respectively.

The result is computed as follows:

\[
d = \min\left(e^l, e^r\right),
\]

\[
c_k = c^l_{p(k)} \odot c^r_{p(k)}, \ k < d,
\]

where \(\odot\) is the operation determined by \(<\text{binary}>\).

A.6.14 choice

This function performs the data-parallel choice and is applicable both to the alternatives of any sort.

\[
\text{CHOICE} ::= ( \text{choice} \text{ <mask> EXPR.s <mask> EXPR.t <mask> EXPR.f } )
\]

Operand \(\text{EXPR.} \text{s}\) must be a logical value, and \(\text{EXPR.} \text{t}\) and \(\text{EXPR.} \text{f}\) must be of the same sort.

Their a posteriori types are determined as follows:

\[
\hat{t} = \hat{t'} = \begin{cases} 
\max(t^l, t^l') & \text{for values} \\
\min(t^l, t^l') & \text{for names and targets}
\end{cases}
\]

The result sort coincides with that of \(\text{EXPR.} \text{t}\) and \(\text{EXPR.} \text{f}\).

\[
d = \min(e^l, e^l, e^r),
\]

\[
c_k = \begin{cases} 
c^l_{p(k)} & \text{if } c^r_{p(k)} = \text{true} \\
c^r_{p(k)} & \text{otherwise}
\end{cases}, \ k < d.
\]

A.6.15 reduce

This function reduces its operand by applying an associative commutative binary operation.

\[
\text{REDUCE} ::= ( \text{reduce} \text{ <total> <mask> EXPR.a } )
\]

The operand must be a nonscalar value. Its a posteriori type is this:

\[
\hat{t}^a = \min\left(t^s, \max\left(t^a, t^a\right)\right),
\]

where \(t^s\) and \(t^a\) are the senior and junior admissible types of the operator \(<\text{total}>\), respectively.

The result is computed as follows:
A.6.16 transp

This function transposes its operand.

\[
\text{TRANSP} := ( \text{transp} \ <\text{number}> \ \text{EXPR}.a )
\]

Operand EXPR.a is an arbitrary object having the rank greater then unity; its a posteriori type coincides with the a priori one. The result sort is the same as that of EXPR.a. Let us define a mapping

\[
T : (a, N) \to b
\]

such that

\[
b_i = \begin{cases} 
    a_i, & 0 \leq i < N \\ 
    a_{i+1}, & N \leq i < r(a) - 1 \\ 
    a_{N}, & i = r(a) - 1 
\end{cases}
\]

Then the shape and contents of the result are determined as follows (\(N\) is the number expressed by the literal):

\[
d = T(d^a, N),
\]

\[
c_k = cT(k, N), \quad k < d.
\]

A.6.17 sect

This function returns a layer of its first operand computed by fixing one of the components of the multi-index.

\[
\text{SECT} := ( \text{sect} \ <\text{number}> \ \text{EXPR}.s \ \text{EXPR}.i )
\]

Operand EXPR.s is a nonscalar of an arbitrary sort. Its a posteriori type coincides with its a priori one. Operand EXPR.i is an aposteriori integer scalar value.

The result sort are the same as that of EXPR.s. This function first turns the number expressed by the literal (which we shall denote as \(N\)) into an auxiliary mask \(m\):

\[
m_j = \begin{cases} 
1, & \text{if } j \neq N \\ 
0, & \text{if } j = N 
\end{cases}, \quad 0 \leq j < r(d^*).
\]

Then the shape and contents of the result are computed as follows:

\[
d = P(d^a, m),
\]

\[
c_k = c^*_l(k), \quad k < d,
\]

where

\[
l_j(k) = \begin{cases} 
    E_j(k, m), & \text{if } j \neq N \\ 
    c^*, & \text{if } j = N 
\end{cases}, \quad 0 \leq j < r(d^*).
\]
A.6. FUNCTIONS

A.6.18 slice

This function slices a regular fragment from its first operand.

SLICE ::= ( slice <number> EXPR.s EXPR.i )

Operand EXPR.s is a nonscalar of an arbitrary sort; its a posteriori type coincides with its a priori one.

Operand EXPR.i is an a posteriori integer one-dimensional value.

The result sort is the same as that of EXPR.s. Its shape and contents are determined as follows (N is the number expressed by <number>):

\[ d_j = \begin{cases} d_j^s, & \text{if } j \neq N \\ d_j^p, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^s), \]

\[ c_k = d_i(k), \quad k < d, \]

where

\[ l_j(k) = \begin{cases} k_i, & \text{if } j \neq N \\ c_{i,j}^k, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^p). \]

A.6.19 repl

This function replicates one of its operands.

REPL ::= ( repl <mask> EXPR.s EXPR.r ... )

Operand EXPR.s can be of any sort, type and rank. Operands EXPR.r, 0 \leq i < n, are scalar values of a posteriori integer type, whose number n must coincide with the number of bits in the mask. The result sort coincides with the sort of EXPR.s.

\[ d_i = \begin{cases} c_i^r, & \text{if } e_i^r = \infty \\ e_i^p c_i^r, & \text{otherwise} \end{cases}, \quad 0 \leq i < n, \]

\[ c_k = c_i^p (k) \pmod{d^r}, \quad k < d. \]

A.6.20 pack

This function repacks its first operand into a different shape array.

PACK ::= ( pack EXPR.s EXPR.e ... )

The first operand is an arbitrary nonscalar. The function returns an object of the same sort as that of EXPR.s. Operands EXPR.e, 0 \leq i < n, are scalar a posteriori integer values.

The result shape and contents are determined as follows:

\[ d_i = e_i^s, \quad 0 \leq i < n, \]

\[ c_k = c_i^l(k), \quad k < d, \]

where l(k) is the multi-index that provides the same sequential number (in terms of lexicographical ordering of indices while flattening an array) in the operand elements as multi-index k in the result.
A.6.21 gather

This function gathers the first operand elements using the logical object provided by the second operand.

\[ \text{GATHER} ::= (\text{gather} <\text{mask}> \text{EXPR.s} \text{EXPR.m}) \]

Operand \text{EXPR.s} is an arbitrary nonscalar object. Operand \text{EXPR.m} is a logical value.

Let us introduce the following denotations: \( \text{A}(v, x) \) is vector \( v \) with the component \( x \) appended; \( \text{R}(v) \) is vector \( v \) with the last component removed; and \( \text{L}(v) \) is the last component of vector \( v: \text{A}(\text{R}(v), \text{L}(v)) \equiv v. \)

The result sort coincides with that of \text{EXPR.s}. The result shape and contents are determined as follows:

\[
d = \text{A} \left( \bar{p}^{\prime}(d^a), \sum_{k < d^m} I(c_k) \right),
\]

where

\[
I(x) = \begin{cases} 1, & \text{if } x = \text{true} \\ 0, & \text{if } x = \text{false} \end{cases};
\]

\[
c_k = G(s^a(\text{R}(k)), e^m)_{L(k)}, \ k < d,
\]

where \( G(A, B) \) is a vector computed by compression of array \( A \) using logical array \( B \) of the same rank. This vector is composed by elements

\[
\{ A_k | k < \min(d^a, d^B) : B_k = \text{true} \},
\]

ordered according to the lexicographical order of their coordinates.

A.6.22 diag

This function slices a hyperdiagonal from its nonscalar operand.

\[ \text{DIAG} ::= (\text{diag} <\text{mask}> \text{EXPR.a}) \]

The only operand \text{EXPR.a} is an arbitrary nonscalar object; its a posteriori type coincides with the a priori one.

The result sort is the same as that of \text{EXPR.a}. Its shape and contents are as follows:

\[
d = \text{A} \left( \bar{p}^{\prime}(d^a), \min_{0 \leq i < n} p_i^a(d^e) \right),
\]

where \( n = r(p^a(d^a)) \) is the number of unity bits in the mask;

\[
c_k = s^a(\text{R}(k))_{q(k)}, \ k < d,
\]

where

\[
q_i(k) = L(k), \ 0 \leq i < n.
\]
A.6. FUNCTIONS

A.6.23 transform

This is a nonscalar analogue of indexing.

TRANSFORM ::= ( transform EXPR.s {<mask> EXPR.t}... )

Operand EXPR.s is an arbitrary nonscalar. All of EXPR.t[i, 0 ≤ i < n, are a posteriori integer values. An equality \( n = r(d) \) must be satisfied.

The result sort is same as the one of EXPR.s.

\[
\begin{align*}
\mathbf{d} &= \min_{0 \leq i < n} e_i, \\
\mathbf{c}_k &= c_{i}(k), \quad k \prec \mathbf{d},
\end{align*}
\]

where

\[
\begin{align*}
l_{i}(k) &= c_{i}(k), \quad 0 \leq i < n.
\end{align*}
\]

A.6.24 part

This function extracts the real or imaginary part from its operand.

PART ::= ( part <selector> EXPR )

The operand may have any sort and is of the a posteriori complex type. The result type is real. Its sort and shape are the same as those of the operand. The contents of the result are determined as follows:

1. If the operand is a value then the result is computed by element-wise extraction of the real or imaginary part of the operand depending on the selector.
2. If the operand is a name or a target then each element of the result will refer to the real or imaginary part of the elementary value referred to by the corresponding (in terms of multi-index) element of the operand. If a target operand element contains the dummy reference, then the corresponding element of the result will contain the dummy reference, too.

A.6.25 comp

This function composes its operands to form a single object.

COMP ::= ( comp <number> <mask> EXPR.l <mask> EXPR.r )

The operands must be of the same sort. An equality \( r(e_l) = r(e_r) \) must be satisfied. The a posteriori types of the operands are as follows:

\[
\begin{align*}
t_l &= t_r, \\
t_l &= t_r = \{i_l, t_r\} & \text{for values} \\
\end{align*}
\]
APPENDIX A. DEFINITION OF F-CODE

The result sort coincides with that of EXP.R.1 and EXP.R.r.

Let \( N \) be the number expressed by \(<\text{number}>\) and

\[
g^x = \begin{cases} 1, & \text{if } \frac{\text{r}}{\text{c}} = \infty \\ \text{ otherwise } & \end{cases}
\]

Now the shape and contents of the result are as follows

\[
d_j = \begin{cases} \min(e_j, e'_j), & \text{if } j \neq N \\ e_j + g', & \text{if } j = N, \ 0 \leq j < r(e') \end{cases},
\]

\[
c_k = \begin{cases} c^p_{\text{pr}(k)} & \text{if } k \text{ is } \text{r}^\text{e} \\ \text{ otherwise } & \end{cases}, \quad k < d,
\]

where

\[
q_i(k) = \begin{cases} k_i, & i \neq N \\ k_i - g', & i = N, \ 0 \leq i < r(d). \end{cases}
\]

A.6.26 pol

This is evaluation of a polynomial of an arbitrary number of (nonscalar) variables.

\[
\text{POL} ::= ( \text{pol exp.r} \{<\text{mask}> \text{EXPR.r}\} \ldots )
\]

Operand \text{EXPR.r} is a nonscalar value. All of \text{EXPR.r}, \( 0 \leq i < n \), are values. An equality \( n = r(d^e) \) must be satisfied. The a posteriori types of the operands are as follows:

\[
c^e - t^p_0 = \ldots = t^p_{n-1} = \max(t^e, t^p_0, \ldots , t^p_{n-1}, \text{integer})
\]

The result is computed as follows:

\[
d = \min_{0 \leq i < n} e^v_i,
\]

\[
c_k = \sum_{1 \leq d^e} \prod_{0 \leq i < n} \left( c^v_{\text{pr}(k)} \right)^{l_i}, \quad k < d.
\]

A.6.27 displace

This function displaces a pointer by some number of steps.

\[
\text{DISPLACE} ::= ( \text{displace MODE} <\text{mask}> \text{EXPR.p} <\text{mask}> \text{EXPR.s} )
\]

\[
\text{MODE} ::= <\text{type}> \mid <\text{identifier}>
\]

Both operands must be values of a posteriori integer type. If an identifier is specified as a mode then it must be associated with a template.

The result is computed as follows:

\[
d = \min(e^p(e'^p), e')
\]

\[
c_k = c^p_{\text{pr}(k)} + M c^z_{\text{pr}(k)}, \quad k < d,
\]

where \( M \) is the size in conventional units of memory of an object of the type specified (if type alternative is chosen as a mode) or of the template specified (if identifier alternative is chosen).
A.6.28 distance

This function evaluates a distance between two pointers in steps.

\[
\text{DISTANCE} ::= (\text{distance}\ \text{MODE}\ <\text{mask}>\ \text{EXPR}.l\ <\text{mask}>\ \text{EXPR}.r)
\]

\[
\text{MODE} ::= \text{<type>} | \text{<identifier>}
\]

Both operands must be values of a posteriori integer type. If an identifier is specified as a mode then it must be associated with a template.

The result is computed as follows:

\[
d = \min(e^l, e^r),
\]

\[
c_k = (c'^l(k) - c'^r(k))/M, \ k < d,
\]

where \( M \) is defined in the same way as in function DISPLACE.

A.6.29 type

This function analyzes the type of its operand.

\[
\text{TYPE} ::= (\text{type}\ \text{EXPR})
\]

The operand is an arbitrary object.

The result of this function is an integer scalar value determined by the operand type as follows:

\[
\begin{align*}
\text{logical} & \quad 1 \\
\text{character} & \quad 2 \\
\text{integer} & \quad 3 \\
\text{real} & \quad 4 \\
\text{complex} & \quad 5
\end{align*}
\]

A.6.30 shape

This function returns the shape of its operand.

\[
\text{SHAPE} ::= (\text{shape}\ \text{EXPR}.a)
\]

The operand is an arbitrary object.

\[
d_0 = r(d^c),
\]

\[
c_i = d_i^c, \ 0 \leq i < d_0.
\]
A.6.31 assign

This function always returns an integer scalar value 0. It is used to achieve a side effect, which is changing the contents of the values referred to by some name or target.

 ASSIGN ::= ( assign EXPR.l <mask> EXPR.r )

Operand EXPR.l must be of sort name or target, and EXPR.r of sort value. Their a posteriori types coincide with the a priori ones. The action undertaken by the function is:

\[ c^k \leftarrow c^p(k), \quad k \prec d : p^r(k) \prec d^r, \]

where \( a \leftarrow b \) means the following. If \( a \) is a dummy reference then there is nothing to do, otherwise the contents of the scalar value referred to by \( a \) is replaced by \( b \) coerced to the type of that value.

A.6.32 channel

This function creates a channel and arranges a scope for it. It does not assume its argument to be precomputed.

 CHANNEL ::= ( channel <identifier> EXPR.a <sort> <type> <number> )

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.

2. The identifier is associated with a channel being a FIFO queue of primitive objects. Type, sort and rank specified are ascribed to the channel (the latter characteristic being defined by <number>). On creation, the channel is empty.

3. The argument EXPR.a is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.

4. Any objects that may remain in the channel are disposed.

5. The current association for the identifier is destroyed and the saved one (if any) is restored.

6. Operand EXPR.a is returned.

A.6.33 put

This function puts an object into a channel.

 PUT ::= ( put <identifier> EXPR )

The identifier specified must be associated with a channel. The type ascribed to the channel determines the a posteriori type of the operand. Sort and rank of the operand must coincide with those ascribed to the channel. This function yields its operand as a result.
A.6.34 get

This function gets an object from a channel and returns it as the result.

\[
\text{GET} ::= (\ \text{get } \text{<identifier> } )
\]

The identifier specified must be associated with a channel. If the channel is empty then the function evaluation is suspended until an object appears in the channel.

A.6.35 seq

This function sequentially computes its arguments.

\[
\text{SEQ} ::= (\ \text{seq } \text{EXPR } ... )
\]

The arguments are evaluated sequentially from left to right and must return a posteriori integer scalar values. If current operand is 0, the next one (if any) is selected. If the current operand is greater than 0, the function terminates returning the current operand value minus one. If all the operands have happened to be zeroes it returns zero as well. This provides for correct completion: any argument can terminate this function by returning 1, or even terminate any enveloping \text{SEQ} or \text{LOOP} (see Section A.6.38) by returning a proper positive value. Finally, if a negative value is returned by any one of the arguments, \text{SEQ} will terminate the F-program.

A.6.36 comma

This function computes its first (left) argument, then computes its second (right) argument, returning the argument defined in direction, and discards the other operand.

\[
\text{<direction>} ::= \text{left } | \text{right}
\]

\[
\text{COMMA} ::= (\ \text{comma } \text{<direction>} \text{EXPR.d EXPR.r})
\]

This function can be used to sequentialize two activities which yield some data object to be processed further, and which for that reason can not be placed in a \text{SEQ} list. When sequentialization is not needed, function \text{CHOICE} can be used.

A.6.37 par

This function serves to parallelize a set of activities. As most of the functions, it assumes its arguments to be precomputed in parallel.

\[
\text{PAR} ::= (\ \text{par } \text{EXPR } ... )
\]

The operands must be scalar values of a posteriori integer type. This function is introduced to support parallel evaluation of side-effecting functions. If all the operands are nonnegative, the function returns the maximum of these, otherwise it returns the minimum operand. This is performed for correct interpretation of completions.
A.6.38 loop

This function repeatedly computes its argument.

\[
\text{LOOP ::= ( loop EXPR )}
\]

The result returned by the argument every time it is being computed must be an a posteriori integer scalar value. If it is equal to zero, the loop continues. If it is positive, the loop terminates returning this value minus one. A negative integer returned by the argument will cause termination of the F-program.

A.6.39 spawn

This function executes a number of identical activities in parallel. It does not assume its last argument to be precomputed.

\[
\text{SPAWN ::= ( spawn <identifier> EXPR.a EXPR.n )}
\]

The operand \( \text{EXPR.n} \) must be a positive, a posteriori integer, scalar value (let us denote it as \( N \)). The algorithm of function evaluation is this:

1. The existing association for the identifier is saved.

2. \( N \) integer scalar values \( 1, \ldots, N \) are created.

3. The argument \( \text{EXPR.a} \) is computed \( N \) times in parallel, the identifier being associated with one (unique) of the values created at the previous step prior to each evaluation of the argument. All the results yielded must be a posteriori integer scalar values.

4. The saved associations for the identifier (if any) is restored.

5. If all the results yielded at the step 3 are nonnegative, the function returns the maximum of these, otherwise it returns the minimum. This is for correct interpretation of completions.

A.6.40 if

This is a usual if-then-else construct. It does not assume the last two arguments to be precomputed.

\[
\text{IF ::= ( if EXPR.c EXPR.t EXPR.f )}
\]

Operand \( \text{EXPR.c} \) must be a logical scalar value. If it is \textit{true} then \( \text{EXPR.t} \) is computed, otherwise \( \text{EXPR.f} \). In either case the operand computed must be an integer scalar value, which is then returned.
A.6. FUNCTIONS

A.6.41 call

This function invokes the subroutine pointed to by its operand (which must be a posteriori integer scalar value) and returns its result.

CALL ::= ( call EXPR <type> <sort> <number> )

<type>, <sort>, <number> must coincide with the type, sort, and rank of the object returned by an instance of call.
Lexical table

Lexical analysis turns a stream of characters into a stream of tokens. Tokens in this case are values being indices into the lexical table:

```
lex_table
/*
lexer[] (NAME) (VALUE) (CLASS)
{ "(" , OPEN, LEX },
{ ")" , CLOSE, LEX },
{ "const" , CONST, FUNCTION },
{ "hold" , HOLD, FUNCTION },
{ "template" , TEMPLATE, FUNCTION },
{ "create" , CREATE, FUNCTION },
{ "var" , VAR, FUNCTION },
{ "select" , SELECT, FUNCTION },
{ "global" , GLOBAL, FUNCTION },
{ "dispose" , DISPOSE, FUNCTION },
{ "local" , LOCAL, FUNCTION },
{ "mark" , MARK, FUNCTION },
{ "ramp" , HAMP, FUNCTION },
{ "monadic" , MONADIC, FUNCTION },
{ "dyadic" , DYADIC, FUNCTION },
{ "choice" , CHOICE, FUNCTION },
{ "reduce" , REDUCE, FUNCTION },
{ "transp" , TRANS, FUNCTION },
{ "sect" , SECT, FUNCTION },
{ "slice" , SLICE, FUNCTION },
{ "repl" , REPL, FUNCTION },
{ "pack" , PACK, FUNCTION },
{ "gather" , GATHER, FUNCTION },
{ "diag" , DIAG, FUNCTION },
{ "transform" , TRANSFORM, FUNCTION },
{ "part" , PART, FUNCTION },
{ "comp" , COMP, FUNCTION },
{ "pol" , POL, FUNCTION },
{ "displace" , DISPLACE, FUNCTION },
{ "distance" , DISTANCE, FUNCTION },
{ "type" , TYPE_FUNC, FUNCTION },
{ "shape" , SHAPE, FUNCTION },
{ "assign" , ASSIGN, FUNCTION },
{ "channel" , CHANNEL, FUNCTION },
{ "put" , PUT, FUNCTION },
{ "get" , GET, FUNCTION },
{ "seq" , SEQ, FUNCTION },
{ "comma" , COMMA, FUNCTION },
{ "par" , PAR, FUNCTION }
*/
```

233
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;loop&quot;</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>&quot;spawn&quot;</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>&quot;if&quot;</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>&quot;call&quot;</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>&quot;neg&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;inv&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;frac&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;sqrt&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;arg&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;conj&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;modulus&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;sin&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;cos&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;tan&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;arcsin&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;arccos&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;arctan&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;sinh&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;cosh&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;tanh&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;exp&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;ln&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;even&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;odd&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;not&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;bit-not&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;round&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;trunc&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;font&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;ref&quot;</td>
<td>UNARY</td>
</tr>
<tr>
<td>&quot;add&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;mul&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;max&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;min&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;and&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;or&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;xor&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;bit-and&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;bit-or&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;bit-xor&quot;</td>
<td>TOTAL BINARY</td>
</tr>
<tr>
<td>&quot;sub&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;div&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;idiv&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;modulo&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;pow&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;ipow&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;log&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;shr&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;shl&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;ror&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;rol&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;gt&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;ge&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;lt&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;le&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;eq&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;ne&quot;</td>
<td>BINARY</td>
</tr>
<tr>
<td>&quot;logical&quot;</td>
<td>TYPE</td>
</tr>
<tr>
<td>&quot;character&quot;</td>
<td>TYPE</td>
</tr>
<tr>
<td>&quot;integer&quot;</td>
<td>TYPE</td>
</tr>
<tr>
<td>&quot;real&quot;</td>
<td>TYPE</td>
</tr>
<tr>
<td>&quot;complex&quot;</td>
<td>TYPE</td>
</tr>
</tbody>
</table>
The token $i$ is an index into this table. $T_N(i)$ is the string (useful both for lexical analysis and pretty-printing), $T_C(i)$ is the token class (important for parsing), $T_V(i)$ is the token value (an enumeration of a particular class).
Appendix C

Parse table

<table>
<thead>
<tr>
<th>t.expr</th>
<th>LEX, OPEN, t.funcs, t.close</th>
</tr>
</thead>
<tbody>
<tr>
<td>t.selector</td>
<td>SELECTOR, RIGHT, t.expr, t.expr</td>
</tr>
<tr>
<td>t.select</td>
<td>SELECTOR, t.expr, t.expr</td>
</tr>
<tr>
<td>t.int</td>
<td>INT, t.expr, t.expr</td>
</tr>
<tr>
<td>t.int.expr</td>
<td>INT, t.expr, t.expr</td>
</tr>
<tr>
<td>t.cid</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.id</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.vid</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.vid.expr</td>
<td>IDENTIFIER, t.id, t.expr</td>
</tr>
<tr>
<td>t.id.expr</td>
<td>IDENTIFIER, t.id, t.expr</td>
</tr>
<tr>
<td>t.tid</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.id.tid</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.id.c.expr</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.select</td>
<td>SORT, t.tid, t.expr</td>
</tr>
<tr>
<td>t.select</td>
<td>ACCESS, POINTER, t.tid, t.expr</td>
</tr>
<tr>
<td>t.acc.id</td>
<td>ACCESS, t.vid, t.expr</td>
</tr>
<tr>
<td>t.check</td>
<td>ACCESS, t.vid, t.expr</td>
</tr>
<tr>
<td>t.type</td>
<td>TYPE, t.type, t.vid</td>
</tr>
<tr>
<td>t.stn</td>
<td>SORT, t.type, t.vid</td>
</tr>
<tr>
<td>t.ptn</td>
<td>PROPERTY, t.type, t.vid</td>
</tr>
<tr>
<td>t.close</td>
<td>LEX, CLOSE, t.vid, t.close</td>
</tr>
<tr>
<td>t.expr2m</td>
<td>LEX, t.expr, t.expr</td>
</tr>
<tr>
<td>t.expr2</td>
<td>LEX, t.expr, t.expr</td>
</tr>
<tr>
<td>t.expres</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.expres</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.expres</td>
<td>TEMP</td>
</tr>
<tr>
<td>t.expres</td>
<td>LEX, OPEN, t.vid, t.expr, t.expres</td>
</tr>
<tr>
<td>t.mask.expres</td>
<td>MASK, t.vid, t.expr</td>
</tr>
<tr>
<td>t.mask.expres</td>
<td>LEX, t.vid, t.expr</td>
</tr>
<tr>
<td>t.mask.expres</td>
<td>T.MASK, t.vid, t.expres</td>
</tr>
<tr>
<td>t.mask.expres</td>
<td>LEX, OPEN, t.vid, t.expres</td>
</tr>
<tr>
<td>t.mode</td>
<td>TYPE, t.mode, t.vid</td>
</tr>
<tr>
<td>t.mode</td>
<td>IDENTIFIER, T.TEMPLATE, t.vid</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>TEMP, LEX, CLOSE, t.vid, t.mode</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>TEMP, TYPE, t.mode, t.mode</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>TEMP, IDENTIFIER, t.mode, t.mode</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>FUNCTION, SEQ, t.mode, t.mode</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>FUNCTION, TRANSP, t.mode, t.mode</td>
</tr>
<tr>
<td>t.mode.expres</td>
<td>FUNCTION, CONST, t.mode, t.mode</td>
</tr>
</tbody>
</table>

237
APPENDIX C. PARSE TABLE

t_funcs ( FUNCTION, VAR, t_acc_id, ⊥ )
t_funcs ( FUNCTION, COMMA, t_selector, ⊥ )
t_funcs ( FUNCTION, ASSIGN, t_expr, t_mask_expr )
t_funcs ( FUNCTION, COMP, t_int, t_mask_expr2 )
t_funcs ( FUNCTION, HOLD, t_id_expr, t_expr )
t_funcs ( FUNCTION, TEMPLATE, t_id, t_expr, t_mode_exprs )
t_funcs ( FUNCTION, CREATE, t_id, t_mode_expr )
t_funcs ( FUNCTION, SELECT, t_selector, t_int_expr )
t_funcs ( FUNCTION, GLOBAL, t_mode, t_exprs )
t_funcs ( FUNCTION, DISPOSE, t_expr, ⊥ )
t_funcs ( FUNCTION, LOCAL, t_mode, t_exprs )
t_funcs ( FUNCTION, MARK, t_expr, ⊥ )
t_funcs ( FUNCTION, RAMP, t_expr2, t_expr )
t_funcs ( FUNCTION, MONADIC, t_monadic, ⊥ )
t_funcs ( FUNCTION, DYADIC, t_dyadic, ⊥ )
t_funcs ( FUNCTION, CHOICE, t_mask_expr, t_mask_expr2 )
t_funcs ( FUNCTION, REDUCE, t_reduce, ⊥ )
t_funcs ( FUNCTION, SECT, t_int_expr, t_expr )
t_funcs ( FUNCTION, SLICE, t_int_expr, t_expr )
t_funcs ( FUNCTION, REPL, t_mask_expr, t_exprs )
t_funcs ( FUNCTION, PACK, t_expr, t_exprs )
t_funcs ( FUNCTION, GATHER, t_mask_expr, t_expr )
t_funcs ( FUNCTION, DIAG, t_mask_expr, ⊥ )
t_funcs ( FUNCTION, TRANSFORM, t_expr, t_mask_exprs )
t_funcs ( FUNCTION, PART, t_sel_expr, ⊥ )
t_funcs ( FUNCTION, POL, t_expr, t_mask_exprs )
t_funcs ( FUNCTION, DISPLACE, t_mode, t_mask_expr2 )
t_funcs ( FUNCTION, DISTANCE, t_mode, t_mask_expr2 )
t_funcs ( FUNCTION, TYPE_FUNC, t_expr, ⊥ )
t_funcs ( FUNCTION, SHAPE, t_expr, ⊥ )
t_funcs ( FUNCTION, CHANNEL, t_id_c_expr, t_stn )
t_funcs ( FUNCTION, PUT, t_cid, t_expr )
t_funcs ( FUNCTION, GET, t_cid, _ )
t_funcs ( FUNCTION, PAR, t_expr, t_exprs )
t_funcs ( FUNCTION, LOOP, t_expr, ⊥ )
t_funcs ( FUNCTION, SPAWN, t_vid_expr, t_expr )
t_funcs ( FUNCTION, IF, t_expr, t_expr2 )
t_funcs ( FUNCTION, CALL, t_id, t_stn )
t_funcs ( FUNCTION, COERCED, t_id_expr, t_ptn )
t_const ( LEX, OPEN, tfuncs, t_close )
t_const ( LEXCONST, ⊥, ⊥, ⊥ )
t_const ( NUMBER, ⊥, ⊥, ⊥ )
t_monadic ( UNARY, REF, t_expr, ⊥ )
t_monadic ( UNARY, MODULUS, t_expr, ⊥ )
t_monadic ( UNARY, NEG, t_expr, ⊥ )
t_monadic ( UNARY, INV, t_expr, ⊥ )
t_monadic ( UNARY, FRAC, t_expr, ⊥ )
t_monadic ( UNARY, SQRT, t_expr, ⊥ )
t_monadic ( UNARY, ARG, t_expr, ⊥ )
t_monadic ( UNARY, CONJ, t_expr, ⊥ )
t_monadic ( UNARY, SIN, t_expr, ⊥ )
t_monadic ( UNARY, COS, t_expr, ⊥ )
t_monadic ( UNARY, TAN, t_expr, ⊥ )
t_monadic ( UNARY, ARCSIN, t_expr, ⊥ )
t_monadic ( UNARY, ARCCOS, t_expr, ⊥ )
t_monadic ( UNARY, ARCTAN, t_expr, ⊥ )
t_monadic ( UNARY, SINH, t_expr, ⊥ )
t_monadic ( UNARY, COSH, t_expr, ⊥ )
t_monadic ( UNARY, TANH, t_expr, ⊥ )
t_monadic ( UNARY, EXP, t_expr, ⊥ )
t_monadic ( UNARY, LN, t_expr, ⊥ )
t_monadic ( UNARY, EVEN, t_expr, ⊥ )
t_monadic ( UNARY, ODD, t_expr, ⊥ )
t_monadic ( UNARY, NOT, t_expr, ⊥ )
t_monadic ( UNARY, BIT_NOT, t_expr, ⊥ )
t_monadic ( UNARY, ROUND, t_expr, ⊥ )
t_monadic ( UNARY, TRUNC, t_expr, ⊥ )
t_monadic ( UNARY, FONT, t_expr, ⊥ )
t_monadic ( BINARY, MIN, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, ADD, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, SUB, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, DIV, t_mask_expr, t_mask_expr )
t_monadic ( BINARY,_MUL, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, MAX, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, AND, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, OR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, XOR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, BIT_AND, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, BIT_OR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, BIT_XOR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, IDIV, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, MODULO, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, IPOW, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, POW, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, LOG, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, SHR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, SHL, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, ROR, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, ROL, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, GT, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, GE, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, LT, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, LE, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, EQ, t_mask_expr, t_mask_expr )
t_monadic ( BINARY, NE, t_mask_expr, t_mask_expr )
t_reduce ( TOTAL, ADD, t_mask_expr, ⊥ )
t_reduce ( TOTAL, MUL, t_mask_expr, ⊥ )
t_reduce ( TOTAL, MAX, t_mask_expr, ⊥ )
t_reduce ( TOTAL, AND, t_mask_expr, ⊥ )
t_reduce ( TOTAL, OR, t_mask_expr, ⊥ )
t_reduce ( TOTAL, XOR, t_mask_expr, ⊥ )
t_reduce ( TOTAL, BIT_AND, t_mask_expr, ⊥ )
t_reduce ( TOTAL, BIT_OR, t_mask_expr, ⊥ )
t_reduce ( TOTAL, BIT_XOR, t_mask_expr, ⊥ )
t_mask_expr ( MASK, ⊥, t_expr, ⊥ )
t_mask_expr ( TEMP | LEX, OPEN, t_expr, ⊥ )
## Appendix D

### Type inference table

<table>
<thead>
<tr>
<th>Type expression</th>
<th>Description</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_expr (LEX, OPEN, t_func, t_close, (-j-&gt;~))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_selector (SELECTOR, RIGHT, t_expr, t_expr, (Rt,Rr, Rs), (-,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_selector (SELECTOR, LEFT, t_expr, t_expr, (Lt,Lr,Ls), (-,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_close (LEX, CLOSE, X, X, (<em>,-,-), (</em>,<em>,</em>))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_int (INT, X, X, (<em>,T,</em>), (Int,-,-), (Int,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_cid (TEMP</td>
<td>IDENTIFIER, T_CHANNEL, t_id, (-,-), (Int,-,-), (Int,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_tid (TEMP</td>
<td>IDENTIFIER, T_TEMPLATE, t_id, X, (Int,-,-), (Int,-,-), (Int,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_vid (TEMP</td>
<td>IDENTIFIER, T_VAR, t_id, X, (-,-), (-,-), (-,-))</td>
<td></td>
</tr>
<tr>
<td>t_id (IDENTIFIER, X, X, (Int,-,-), (Int,-,-), (Int,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_check (X, t_expr, X, (-,-), (-,-), (-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>r_check (-, t_expr, X, (-,-), (-,-), (-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>s_check (-, t_expr, X, (-,-), (-,-), (-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_type (TYPE, X, X, (T,-,-), (T,-,-), (T,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_stn (SORT, t_type, taint, (Lt,Rr,T), (-,-), (-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_ptn (PROPERTY, t_type, t_int, (Lt,Lr,T), (Int,-,-), (Int,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_mask_expr (MASK, t_expr, X, (Lt,W,Ls), (Mt), (Mt), (Mt))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_mask_expr (TEMP</td>
<td>MASK, t_expr, t_expr, (St,Sr,Ss), (Mt), (Mt), (Mt))</td>
<td></td>
</tr>
<tr>
<td>t_exprs (TEMP</td>
<td>LEX, OPEN, t_expr, t_expr, (-,-), (-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr2m (-, t_expr, t_expr, (-,-), (Mt), (Mt))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr2 (-, t_expr, t_expr, (-,-), (-,-,-), (-,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_expr2m (-, t_expr, t_expr, (-,-), (-,-,-), (-,-,-))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_sh (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (TEMP</td>
<td>TYPE, X, X, (T,-,-), (T,-,-), (T,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, OPEN, t_expr, t_expr, (St,Sr,Ss), (St,Sr,Ss))</td>
<td></td>
</tr>
<tr>
<td>t_mask_expr (MASK, t_expr, t_expr, (Lt,W,Ls), (Mt), (Mt), (Mt))</td>
<td></td>
<td>(-,-,-)</td>
</tr>
<tr>
<td>t_mask_expr (TEMP</td>
<td>LEX, OPEN, t_expr, t_expr, (St,Sr,Ss), (St,Sr,Ss))</td>
<td></td>
</tr>
<tr>
<td>t_exprs (TEMP</td>
<td>T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
</tr>
<tr>
<td>t_expr2 (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (TEMP</td>
<td>TYPE, X, X, (T,-,-), (T,-,-), (T,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
<tr>
<td>t_expr (T</td>
<td>LEX, CLOSE, X, X, (_,-,-), (-,-,-))</td>
<td></td>
</tr>
</tbody>
</table>

241
t_monadic (UNARY, ARCTAN, t_expr, \( (Rea,Lr,Val), (Rea,Val) \), \( (-,-) \))

\[t_monadic \ (UNARY, \ SINH, t_expr, X, (Rea, Lr, Val), (Rea,_,Val), (-,-,-))\]

\[t_monadic \ (UNARY, \ COSH, t_expr, X, (Rea, Lr, Val), (Rea,Val), (-,-,,-))\]

\[t_monadic \ (UNARY, \ TANH, t_expr, X, (Rea, Lr, Val), (Rea,Val), (-,-,,-))\]

\[t_monadic \ (UNARY, \ LN, t_expr, X, (Rea, Lr, Val), (Rea,Val), (-,-,,-))\]

\[t_monadic \ (UNARY, \ EVEN, t_expr, X, (Log, Lr, Val), (Int,Val), (-,-,-))\]

\[t_monadic \ (UNARY, \ ODD, t_expr, X, (Log, Lr, Val), (Int,Val), (-,-,-))\]

\[t_monadic \ (UNARY, \ NOT, t_expr, \ (Log, Lr, Val), (Int,Val), (-,-,,-))\]

\[t_monadic \ (UNARY, \ BIT-NOT, t_expr, X, (Int, Lr, Val), (Int,Val), (-,-,,-))\]

\[t_monadic \ (UNARY, \ ROUND, t_expr, X, (Int, Lr, Val), (Rea,Val), (-,-,-))\]

\[t_monadic \ (UNARY, \ TRUNC, t_expr, X, (Int, Lr, Val), (Rea,Val), (-,-,-))\]

\[t_monadic \ (UNARY, \ FONT, t_expr, X, (Cha, Lr, Val), (Int,-,Val), (-,-,-))\]

\[t_dyadic \ (BINARY, \ MIN, t_mask_expr, t_mask_expr, (5t(Cha|Int|Rea),5r,Val), (Xs(Cha|Int|Rea),Sr,Val), (X,(Cha|Int|Rea),-Sr,Val))\]

\[t_dyadic \ (BINARY, \ ADD, t_mask_expr, t_mask_expr, (S,t(Int|Rea|Com),Sr,Val), (Xs(Int|Rea|Com),Sr,Val))\]

\[t_dyadic \ (BINARY, \ SUB, t_mask_expr, t_mask_expr, (5t(Int|Rea|Com),5r,-,Val), (X,5t(Int|Rea|Com),Sr,Val), (X,(Int|Rea|Com),5r,Val))\]

\[t_dyadic \ (BINARY, \ DIV, t_mask_expr, t_mask_expr, (StfRealCom),Sr,Val), (X,StfRealCom),Sr,Val))\]

\[t_dyadic \ (BINARY, \ MUL, t_mask_expr, t_mask_expr, (S,t(Int|Rea|Com),Sr,Val), (Xs(Int|Rea|Com),Sr,Val))\]

\[t_dyadic \ (BINARY, \ MAX, t_mask_expr, t_mask_expr, (St(Cha|Int|Rea),Sr,Val), (X,(Cha|Int|Rea),5r,Val), (Xs(Cha|Int|Rea),5r,Val))\]

\[t_dyadic \ (BINARY, \ AND, t_mask_expr, t_mask_expr, (Log,Sr,Val), (Log,Sr,Val), (Log,Sr,Val))\]

\[t_dyadic \ (BINARY, \ OR, t_mask_expr, t_mask_expr, (Log,SV>Val), (Log,SV>Val), (Log,SV>Val))\]

\[t_dyadic \ (BINARY, \ XOR, t_mask_expr, t_mask_expr, (Log,SV,Val), (Log,SV,Val), (Log,-SV,Val))\]

\[t_dyadic \ (BINARY, \ BIT-AND, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ BIT-OR, t_mask_expr, t_mask_expr, (Int,SV,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ BIT-XOR, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ IDIV, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ MODULO, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ IPOW, t-mask_expr, t-mask_expr, (Lt,Sr,Val), (Int|Rea|Com,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ POW, t-mask_expr, t-mask_expr, (Rea,Sr,Val), (Rea,Sr,Val), (Rea,Sr,Val))\]

\[t_dyadic \ (BINARY, \ LOG, t_mask_expr, t_mask_expr, (Rea,Sr,Val), (Rea,Sr,Val), (Rea,Sr,Val))\]

\[t_dyadic \ (BINARY, \ SHR, t_mask_expr, t_mask-expr, (Int,-Sr,Val), (Int,-Sr,Val), (Int,-Sr,Val))\]

\[t_dyadic \ (BINARY, \ SHL, t_mask_expr, t_mask_expr, (Int,5Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ ROR, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ ROL, t_mask_expr, t_mask_expr, (Int,Sr,Val), (Int,Sr,Val), (Int,Sr,Val))\]

\[t_dyadic \ (BINARY, \ GT, t_mask_expr, t_mask_expr, (Log,Sr,Val), (Mt(Int,Rea),Sr,Val), (Mt(Int,Rea),Sr,Val))\]

\[t_dyadic \ (BINARY, \ GE, t_mask_expr, t_mask_expr, (Log,Sr,Val), (Mt(Int,Rea),Sr,Val), (Mt(Int,Rea),Sr,Val))\]

\[t_dyadic \ (BINARY, \ LT, t_mask_expr, t_mask_expr, (Log,Rea,Val), (Mt(Int,Rea),Sr,Val), (Mt(Int,Rea),Sr,Val))\]

\[t_dyadic \ (BINARY, \ LE, t_mask_expr, t_mask_expr, (Log,Rea,Val), (Mt(Int,Rea),Sr,Val), (Mt(Int,Rea),Sr,Val))\]

\[t_dyadic \ (BINARY, \ EQ, t_mask_expr, t_mask_expr, (Log,Rea,Val), (Log[Cha|Int|Rea],Sr,Val), (Log[Cha|Int|Rea],Sr,Val))\]

\[t_dyadic \ (BINARY, \ NE, t_mask_expr, t_mask_expr, (Log,Rea,Val), (Log[Cha|Int|Rea],Sr,Val), (Log[Cha|Int|Rea],Sr,Val))\]

\[t_reduce \ (TOTAL, \ ADD, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]

\[t_reduce \ (TOTAL, \ MUL, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]

\[t_reduce \ (TOTAL, \ MAX, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]

\[t_reduce \ (TOTAL, \ AND, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]

\[t_reduce \ (TOTAL, \ OR, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]

\[t_reduce \ (TOTAL, \ XOR, t_mask_expr, \ (E, Int[Real], G, Val), (X, Int[Real], Val))\]
APPENDIX D. TYPE INFERENCE TABLE

\begin{verbatim}
t\_reduce (TOTAL, BIT\_AND, t\_nmask\_expr, ⊥, (Int,G,Val), (Int,⊥,Val), (⊥,⊥))
t\_reduce (TOTAL, BIT\_OR, t\_nmask\_expr, ⊥, (Int,G,Val), (Int,⊥,Val), (⊥,⊥))
t\_reduce (TOTAL, BIT\_XOR, t\_nmask\_expr, ⊥, (Int,G,Val), (Int,⊥,Val), (⊥,⊥))
t\_nmask\_expr (MASK, t\_expr, ⊥, (Lt,X,Ls), (⊥,W,⊥), (⊥,⊥))
t\_nmask\_expr (TEMP \| LEX, OPEN, t\_expr, ⊥, (⊥,0,), (⊥,⊥), (⊥,⊥))
\end{verbatim}
Appendix E

Type inference rules

The parser maintains a pointer in each node of the parse tree to the parse-table entry which created it. See section 3.2. The parse table can be modified to include type inference rules, the rules are then accessible at every node of the parse tree.

Since every node in the tree is a binary one, three sets of rules are included:

- rules : \((U_r, U_l, U_s), (L_r, L_l, L_s), (R_r, R_l, R_s))\)

Where \(U_x\) is the rule denoting an assertion towards the root of the tree, \(L_x\) is the rule denoting an assertion on the left of the node, and \(R_x\) is the rule denoting an assertion on the right of the node. \(A_x\) is the rule for type, \(A_r\) is the rule for rank, and \(A_s\) is the rule for sort.

In cases where no assertion can be made, the symbol \(-\) is used. This, along with a large number of inference rules are all that are needed to make an F-tree monomorphic.

Appendix D contains the modified parse-table which includes type inference rules, and a description of all rules required to successfully make F-code monomorphic, and compilable.

Assertions

All inference rules make some assertion. Assertions can be made both up and down the tree, but each assertion will only made to a single node \(N_i\), and will modify its type-information \(I_i\).

An assertion consists of a modification of the inference as exists \(I_i\) to some new inference \(I'_i\). The three attributes of an inference: type, rank and sort are independent, and so the assertions are independent.

Inference rules take place in one of three positions: \(U_x\), on the node \(N_i\), itself; \(L_x\), on the left child of \(N_i\); \(R_x\), on the right child of \(N_i\). It is obvious, therefore, that implementationally, there needs to be no pointer from a child node to its parent. The tree created by the parser can be singly linked.

Rank assertions

Given \(I_i\), and a rank assertion \(r\). \(\text{rank}'\) becomes \(r\), if \(\text{rank} = \tau \lor \text{rank} = r\). If this is not the case, then there is a rank conflict. This is a fatal error, as far as the program is concerned. The
tree undergoes a rewrite, such that \( N_i' \rightarrow (\text{rank.mismatch } N_i) \). The tree is kept coherent so that the rank mismatch is above the place where the fatal conflict takes place. The inference procedure continues regardless. The conflict is, however, sequestered by the rank_mismatch node and no further rank conflict will occur at this point in the tree.

**Sort assertions**

Given \( I_i \), and a sort assertion \( s \), \( sort' \) becomes \( s \), if \( sort = r \lor sort = s \). If this is not the case, then there is a sort conflict. This is a fatal error, as far as the program is concerned. The tree undergoes a rewrite, such that \( N_i' \rightarrow (\text{sort.mismatch } N_i) \). The tree is kept coherent so that the sort mismatch is above the place where the fatal conflict takes place. The inference procedure continues regardless. The conflict is, however, sequestered by the sort_mismatch node and no further sort conflict will occur at this point in the tree.

**Type assertions**

Given \( I_i \), and a type assertion \( t \), where \( t \) is a subset of the ordered set \( \{ \text{logical, character, integer, real, complex} \} \). \( type' \) becomes \( t \cap type \), if \( t \cap type \neq \emptyset \). If the overlap of the two sets is the empty set, there is a type conflict. This is not necessarily a fatal error, as far as the program is concerned. The tree undergoes a rewrite, such that \( N_i' \rightarrow (\text{type.mismatch } N_i) \). The tree is kept coherent so that the type mismatch is above the place where the conflict took place. The inference procedure continues regardless. The conflict is, however, sequestered by the type_mismatch node and no further type conflict will occur at this point in the tree. It can be called a type conflict because there is a disagreement between two assertions (one which was previously made) regarding the type of a particular node in the tree. The rewrite introduces a further node, which may either be a fatal error, or a type coercion.

**Type coercions**

The type inference procedure introduces type_mismatch nodes. Further on in the process of type inference the following position may be attained: \( N_i = (\text{type.mismatch } N_j) \).

The type inference of \( N_i \) is \( I_i \) and the type inference of \( N_j \) is \( I_j \).

If \( \#(type_i) = 1 \land \#(type_j) = 1 \land sort_i = sort_j \land sort_i \neq r \), the following rewrite may be performed:

\[
N_i = (\text{type.mismatch } N_j) \rightarrow N_i = (\text{type.coerce } N_j), (sort_i = value \land type_i > type_j) \lor (sort_i \neq value \land type_i < type_j).
\]

This basically means that a type_mismatch node can be upgraded into a faultless type_coerce node, if the types of its parent and its child are monomorphic. It is also necessary to check that the direction of the type_coerce is correct in order to maintain a strict type hierarchy. If the sort of the parent and child is value, the coercion is allowed to be upwards (\( > \)); otherwise for
target and name, the coercion is allowed to be downwards (<). If these conditions are not met, the fatal error of type_mismatch is kept in the tree since there is no rewrite and the type checker will note the fault.

Rules for $U_t$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_t$</td>
<td>Infers that the type is the same as the left and right types, as soon as they agree.</td>
</tr>
<tr>
<td>$E_t(RANGE)$</td>
<td>Infers that the type is the same as the left type, as long as it is monomorphic, otherwise the type is inferred to be the range given by RANGE.</td>
</tr>
<tr>
<td>$E_t$</td>
<td>Infers that the type is the same as the left type.</td>
</tr>
<tr>
<td>$L_j$</td>
<td>Infers that the type is the same as the left type.</td>
</tr>
<tr>
<td>$L_t$</td>
<td>Infers that the type is the same as the left type.</td>
</tr>
<tr>
<td>$R_t$</td>
<td>Infers that the type is the same as the right type.</td>
</tr>
<tr>
<td>$T$</td>
<td>Infers that the type is the same as the token value of the node.</td>
</tr>
<tr>
<td>$O_t$</td>
<td>Special inference for modulus.</td>
</tr>
<tr>
<td>$I_t$</td>
<td>For an F-function which creates a variable, the type of the variable is inferred from some information at this node. Secondly, it infers the type of this node is the same as the left type.</td>
</tr>
<tr>
<td>$K_{create}$</td>
<td>This is a specialized form of inherit for the F-function create.</td>
</tr>
<tr>
<td>$K_{select}$</td>
<td>This is a specialized form of inherit for the F-function select.</td>
</tr>
</tbody>
</table>

Rules for $U_r$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_r$</td>
<td>Infers that the rank is the same as the left and right ranks, as soon as they agree.</td>
</tr>
<tr>
<td>$L_r$</td>
<td>Infers that the rank is the same as the left rank.</td>
</tr>
<tr>
<td>$L_{r1}$</td>
<td>Infers that the rank is the same as the left rank + 1.</td>
</tr>
<tr>
<td>$R_r$</td>
<td>Infers that the rank is the same as the right rank.</td>
</tr>
<tr>
<td>$Y$</td>
<td>Infers that the rank is the same as the number of 1s in the mask.</td>
</tr>
<tr>
<td>$W$</td>
<td>Infers that the rank is the same as the length of the mask.</td>
</tr>
<tr>
<td>$X$</td>
<td>Infers that the rank is the same as the number of 0s in the mask.</td>
</tr>
<tr>
<td>$T$</td>
<td>Infers that the rank is the same as the value of the token.</td>
</tr>
<tr>
<td>$G_r$</td>
<td>Inference the rank is the same as the reduced rank (includes a default rank of 0, if there is no mask)</td>
</tr>
<tr>
<td>$I_r$</td>
<td>For an F-function which creates a variable, the rank of the variable is inherited from some information at this node. Secondly, it infers the rank of this node is the same as the left rank.</td>
</tr>
<tr>
<td>$Z_r$</td>
<td>Infers that the rank is the same as the left rank - 1.</td>
</tr>
<tr>
<td>$C$</td>
<td>Counts the number of right operands to be the rank</td>
</tr>
<tr>
<td>$K_{create}$</td>
<td>This is a specialized form of inherit for the F-function create.</td>
</tr>
<tr>
<td>$K_{select}$</td>
<td>This is a specialized form of inherit for the F-function select.</td>
</tr>
<tr>
<td>$E_r$</td>
<td>Infers that the rank is the same as the left rank.</td>
</tr>
<tr>
<td>$number$</td>
<td>A number can be given, which is a direct rank assertion.</td>
</tr>
</tbody>
</table>
APPENDIX E. TYPE INFERRENCE RULES

Rules for $U_s$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_s$</td>
<td>Infers that the sort is the same as the left and right sorts, as soon as they agree.</td>
</tr>
<tr>
<td>$L_s$</td>
<td>Infers that the sort is the same as the left sort.</td>
</tr>
<tr>
<td>$R_s$</td>
<td>Infers that the sort is the same as the right sort.</td>
</tr>
<tr>
<td>$T$</td>
<td>Infers that the sort is the same as the value of the token.</td>
</tr>
<tr>
<td>$I_s$</td>
<td>For an F-function which creates a variable, the sort of the variable is inherited from some information at this node. Secondly, it infers the sort of this node is the same as the left sort.</td>
</tr>
<tr>
<td>$K_{create}$</td>
<td>This is a specialized form of inherit for the F-function create.</td>
</tr>
<tr>
<td>$K_{select}$</td>
<td>This is a specialized form of inherit for the F-function select.</td>
</tr>
<tr>
<td>$E_s$</td>
<td>Infers that the sort is the same as the left sort.</td>
</tr>
<tr>
<td>sort</td>
<td>A sort name can be given which is a direct assertion for sort.</td>
</tr>
</tbody>
</table>

Rules for $L_t / R_t$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_t$</td>
<td>Asserts that the left type and the right type are the same, if either of them is yet known</td>
</tr>
<tr>
<td>$L_t$</td>
<td>Asserts that the right type is the same as the left type</td>
</tr>
<tr>
<td>$M_t(MIN, MAX)$</td>
<td>Asserts that the type of the right and left are of the same type, the minimum of which is MIN and the maximum of which is MAX</td>
</tr>
<tr>
<td>$X_u(RANGE)$</td>
<td>Asserts that the type of the left is from this range for a reduce, coercing up to the minimum type</td>
</tr>
<tr>
<td>$X_s(RANGE)$</td>
<td>Asserts that the type of the left and the right is from this range, coercing up to the minimum type which they can have in common</td>
</tr>
<tr>
<td>type</td>
<td>A type name can be given which is a direct assertion for type.</td>
</tr>
</tbody>
</table>

Rules for $L_r / R_r$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_r$</td>
<td>Asserts that the left rank and the right rank are the same, if either of them is yet known</td>
</tr>
<tr>
<td>$Y$</td>
<td>Asserts that the rank is the same as the length of the mask</td>
</tr>
<tr>
<td>$W$</td>
<td>Asserts that the rank is the same as the number of ones in the mask</td>
</tr>
<tr>
<td>$X$</td>
<td>Asserts that the rank is the same as the number of zeros in the mask</td>
</tr>
<tr>
<td>$L_r$</td>
<td>Asserts that the rank is the same as the left rank</td>
</tr>
<tr>
<td>number</td>
<td>A number can be given which is a direct assertion for rank.</td>
</tr>
</tbody>
</table>

Rules for $L_s / R_s$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Tests that the sort is name or target (just a check — can make no such assertion)</td>
</tr>
<tr>
<td>$S_s$</td>
<td>Asserts that the left sort and the right sort are the same, if either of them is yet known</td>
</tr>
<tr>
<td>$L_s$</td>
<td>Asserts that the sort is the same as the left sort</td>
</tr>
<tr>
<td>sort</td>
<td>A sort name can be given which is a direct assertion for sort</td>
</tr>
</tbody>
</table>

The type inference procedure

Inference is not a recursive but rather a token matching procedure. During parsing a chain is created of all nodes which are created by the parser. Nodes are added to the list, such that those nodes of the tree which are known to be trivial to infer — for example constants, whose attributes are known immediately. Each node has initial type information, and it is the job of the inference procedure to make each node degenerate. See section 3.3.

There are two types of type inference rules (see the tables above). The rules are accessible, since every node of the parse tree maintains a pointer to the table entry which created it.
• Using the DOWN rules in the parse/inference table, inferences about subordinates may be asserted from the inference of a particular node. There are those which apply to the left, and those to the right of every node of the binary tree.

• Using the UP rules in the parse/inference table, it may be possible to evaluate the inference in terms of the inference of its subordinates. (For efficiency, DOWN is done before UP).

• If the node has become degenerate, (and its subordinates are already degenerate), it is unlinked from the inference list, and takes no further part in the inference. If a node is degenerate, but its subordinates are not, it cannot be taken from the list, because on the next iteration, the DOWN stage will still need to be carried out. If a node does not become degenerate it is re-linked at the end of the list.

• Inference continues, round-robin, until either the inference list becomes empty, or an element in the list is visited twice, without there being any successful assertion during that particular iteration.

Since easily inferrable nodes are placed at the head of the inference list, they are immediately taken out into the degenerate list. This speeds up the process.

Since the type-inference procedure re-writes the original tree, this approach is ‘cleaner’ than a recursive algorithm. It is, however, more complicated and may be slower.
Appendix F

Abstract machine instructions

Each instruction has up to three arcs, normally of which one is a destination and two are sources. There are some variants on this: three sources; one destination and one source; one source; etc. The definition also describes a definite type for each arc, which the compiler must adhere to when constructing an instruction graph.

Table F.1: All of these instructions provide load or store operations. The name of the function indicates the type it is applicable to. The abstract machine is capable of reading or writing any type atomically, and only one form of memory addressing is permitted, although s1 and s2 can either be registers (ie. other operations of the permitted type) or constants (of the permitted type).

Table F.2: In order for the compiler to construct a monomorphic graph, the abstract machine includes these atomic type-coercions. Which coerce a value of one type into the value of another. The symbol \( < \) is used to denote a type coercion up the hierarchy, and the symbol \( > \) to denote a type coercion down the hierarchy.

Table F.3: These are a set of machine instructions applicable to integer operations. Each arc must be of type word (integer), and as with all arcs they can be either registers (ie. other integer operations) or constants.

Table F.4: These are a set of floating point (real) machine instructions. Each arc must be of type Float (real), and as with all arcs they can be either registers (ie. other real operations) or constants.

Table F.5: Likewise, these are a set of machine instructions for operations on type complex. Each arc must be of type Complex, and as with all arcs they can be either registers (ie. other Complex operations) or constants. The ability to take real or imaginary components of real numbers was dealt with formerly in the table of type coercions. It is assumed that the abstract machine can handle Complex numbers.
Table F.6: These instructions are used to implement control. Most of these normally have no equivalence with real machine instructions, and can be considered to be macros. It is these instructions which introduce control dependencies explicitly — all other operations can be implemented lazily.

Table F.7: These instructions perform arithmetic tests, and always conceptually leave a logical result in a register. Although it is a principle of some RISC architectures to leave the result of a comparison in a register, some have explicitly a control flag for this purpose instead. In order to utilize a control flag, it requires a certain amount of transformation. In chapters 5 and 6, the implementation of T-code to an i860 is shown in brief, and the i860 requires this transformation. Tests like these can be included in arithmetic operations, where the logical result really is required, or as the first operand of an IF where they provide control — for IF see the table for control operations.

Table F.8: This table shows memory control functions. Atomics can be declared of all of the types. It is assumed that only INT and BYTE can be permitted as immediate-type operands which the abstract machine can represent. The other types are assumed to be stored in a data-segment of a program, and loaded separately, using one of the load instructions. Allocation of memory is done using a special function called alloc which is identical to the standard C library function called calloc. How it is actually implemented is not specified.

The functions ret, fret and cret are used to signify subroutine values in the tree. They can be used for register-coalescing purposes — they signify the points where values need to be in specific registers for a return from a function. The compilation process is intended to be compatible with standard C libraries.
<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLD</td>
<td>(BLD d s1 s2) [s1+s2] = d</td>
<td>loads byte from byte-aligned address s1+s2</td>
</tr>
<tr>
<td>BST</td>
<td>(BST d s1 s2) [s1+s2] = d</td>
<td>stores byte at address byte-aligned address s1+s2</td>
</tr>
<tr>
<td>LD</td>
<td>(LD d s1 s2) [s1+s2] = d</td>
<td>loads word from word-aligned address s1+s2</td>
</tr>
<tr>
<td>ST</td>
<td>(ST d s1 s2) [s1+s2] = d</td>
<td>stores word at word-aligned address s1+s2</td>
</tr>
<tr>
<td>FLD</td>
<td>(FLD d s1 s2) [s1+s2] = d</td>
<td>loads float from float-aligned address s1+s2</td>
</tr>
<tr>
<td>FST</td>
<td>(FST d s1 s2) [s1+s2] = d</td>
<td>stores float at float-aligned address s1+s2</td>
</tr>
<tr>
<td>CLD</td>
<td>(CLD d s1 s2) [s1+s2] = d</td>
<td>loads complex from complex-aligned address s1+s2</td>
</tr>
<tr>
<td>CST</td>
<td>(CST d s1 s2) [s1+s2] = d</td>
<td>stores complex at complex-aligned address s1+s2</td>
</tr>
</tbody>
</table>

Table F.1: Load and store instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCONV</td>
<td>(FCONV d s2) [s2]</td>
<td>integer to floating point conversion</td>
</tr>
<tr>
<td>BYT</td>
<td>(BYT d s2) [s2]</td>
<td>word to byte truncation</td>
</tr>
<tr>
<td>ICONV</td>
<td>(ICONV d s2) [s2]</td>
<td>floating point to integer conversion</td>
</tr>
<tr>
<td>CCONV</td>
<td>(CCONV d s2) [s2]</td>
<td>floating point to complex conversion</td>
</tr>
<tr>
<td>BTOC</td>
<td>(BTOC d s2) [s2]</td>
<td>byte to complex conversion</td>
</tr>
<tr>
<td>BTOF</td>
<td>(BTOF d s2) [s2]</td>
<td>byte to floating point conversion</td>
</tr>
<tr>
<td>ITOC</td>
<td>(ITOC d s2) [s2]</td>
<td>integer to complex conversion</td>
</tr>
<tr>
<td>WRD</td>
<td>(WRD d s2) [s2]</td>
<td>byte to word conversion (sign-extension)</td>
</tr>
<tr>
<td>CIM</td>
<td>(CIM d s2) [s2]</td>
<td>imaginary part of complex selection</td>
</tr>
<tr>
<td>CRE</td>
<td>(CRE d s2) [s2]</td>
<td>real part of complex selection</td>
</tr>
</tbody>
</table>

Table F.2: Type coercions
<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS</td>
<td>(d = (s_1 + s_2))</td>
<td>adds signed integers</td>
</tr>
<tr>
<td>ADDU</td>
<td>(d = (s_1 + s_2))</td>
<td>adds unsigned integers</td>
</tr>
<tr>
<td>AND</td>
<td>(d = (s_1 &amp; s_2))</td>
<td>logical and (executed lazily)</td>
</tr>
<tr>
<td>IAND</td>
<td>(d = (s_1 &amp; s_2))</td>
<td>integer and</td>
</tr>
<tr>
<td>MOV</td>
<td>(d = s_2)</td>
<td>move word</td>
</tr>
<tr>
<td>OR</td>
<td>(d = (s_1</td>
<td>s_2))</td>
</tr>
<tr>
<td>IOR</td>
<td>(d = (s_1</td>
<td>s_2))</td>
</tr>
<tr>
<td>SHL</td>
<td>(d = (s_2 &lt;&lt; s_1))</td>
<td>integer shift left</td>
</tr>
<tr>
<td>SHR</td>
<td>(d = (s_2 &gt;&gt; s_1))</td>
<td>integer shift right</td>
</tr>
<tr>
<td>SHRA</td>
<td>(d = (s_2 &gt;&gt; s_1))</td>
<td>integer shift right arithmetic</td>
</tr>
<tr>
<td>SUBS</td>
<td>(d = (s_1 - s_2))</td>
<td>subtract signed integers</td>
</tr>
<tr>
<td>SUBU</td>
<td>(d = (s_1 - s_2))</td>
<td>subtract unsigned integers</td>
</tr>
<tr>
<td>XOR</td>
<td>(d = (s_1 \oplus s_2))</td>
<td>bit-wise/logical exclusive or</td>
</tr>
<tr>
<td>ROL</td>
<td>(d = (s_2.rol.s_1))</td>
<td>integer rotate left</td>
</tr>
<tr>
<td>ROR</td>
<td>(d = (s_2.ror.s_1))</td>
<td>integer rotate right</td>
</tr>
<tr>
<td>MULS</td>
<td>(d = (s_1 \times s_2))</td>
<td>integer signed multiplication</td>
</tr>
<tr>
<td>MULU</td>
<td>(d = (s_1 \times s_2))</td>
<td>integer unsigned multiplication</td>
</tr>
<tr>
<td>NEG</td>
<td>(d = (-s_2))</td>
<td>integer negation</td>
</tr>
<tr>
<td>ABS</td>
<td>(d =</td>
<td>s_2</td>
</tr>
<tr>
<td>BTN</td>
<td>(d = \sim s_2)</td>
<td>integer bit-wise not</td>
</tr>
<tr>
<td>MAX</td>
<td>(d = \max(s_1,s_2))</td>
<td>integer maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>(d = \min(s_1,s_2))</td>
<td>integer minimum</td>
</tr>
<tr>
<td>DIVU</td>
<td>(d = (s_1 / s_2))</td>
<td>integer unsigned division</td>
</tr>
<tr>
<td>MOD</td>
<td>(d = (s_1 % s_2))</td>
<td>integer modulo</td>
</tr>
<tr>
<td>IPOW</td>
<td>(d = (s_1</td>
<td>s_2))</td>
</tr>
</tbody>
</table>

Table F.3: Word operations
<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>( d = (s_1 + s_2) )</td>
<td>adds floating point numbers Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FMOV</td>
<td>( d = s_2 )</td>
<td>moves floating point number Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FMUL</td>
<td>( d = (s_1 \times s_2) )</td>
<td>multiplies floating point number Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FSUB</td>
<td>( d = (s_1 - s_2) )</td>
<td>subtracts floating point numbers Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FDIV</td>
<td>( d = (s_1 / s_2) )</td>
<td>divides floating point numbers Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FSQR</td>
<td>( d = \sqrt{s_2} )</td>
<td>floating point square root Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FNEG</td>
<td>( d = (-s_2) )</td>
<td>floating point negation Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FINV</td>
<td>( d = (1 / s_2) )</td>
<td>floating point reciprocal Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FRAC</td>
<td>( d = \text{frac}(s_2) )</td>
<td>floating point fraction Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FABS</td>
<td>( d =</td>
<td>s_2</td>
</tr>
<tr>
<td>FSIN</td>
<td>( d = \sin(s_2) )</td>
<td>floating point sine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FCOS</td>
<td>( d = \cos(s_2) )</td>
<td>floating point cosine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FTAN</td>
<td>( d = \tan(s_2) )</td>
<td>floating point tangent Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FASIN</td>
<td>( d = \arcsin(s_2) )</td>
<td>floating point arc-sine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FACOS</td>
<td>( d = \arccos(s_2) )</td>
<td>floating point arc-cosine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FATAN</td>
<td>( d = \arctan(s_2) )</td>
<td>floating point arc-tangent Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FSINH</td>
<td>( d = \sinh(s_2) )</td>
<td>floating point hyperbolic sine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FCOSH</td>
<td>( d = \cosh(s_2) )</td>
<td>floating point hyperbolic cosine Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FTANH</td>
<td>( d = \tanh(s_2) )</td>
<td>floating point hyperbolic tangent Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FEXP</td>
<td>( d = e^{s_2} )</td>
<td>floating point exponential Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FLN</td>
<td>( d = \ln(s_2) )</td>
<td>floating point logarithm base e Float ( \rightarrow ) Float</td>
</tr>
<tr>
<td>FLOG</td>
<td>( d = \log(s_1, s_2) )</td>
<td>floating point logarithm Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FMAX</td>
<td>( d = \max(s_1, s_2) )</td>
<td>floating point maximum Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FMIN</td>
<td>( d = \min(s_1, s_2) )</td>
<td>floating point minimum Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>PPOW</td>
<td>( d = (s_1 \uparrow s_2) )</td>
<td>floating point power Float ( \rightarrow ) Float ( \times ) Float</td>
</tr>
<tr>
<td>FIPOW</td>
<td>( d = (s_1 \uparrow s_2) )</td>
<td>floating point integer power Float ( \rightarrow ) Float ( \times ) Word</td>
</tr>
</tbody>
</table>

Table F.4: Float operations
### APPENDIX F. ABSTRACT MACHINE INSTRUCTIONS

#### Table F.5: Complex operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADD</td>
<td>( CADD d s1 s2 )</td>
<td>complex addition Complex ← Complex × Complex</td>
</tr>
<tr>
<td>CSUB</td>
<td>( CSUB d s1 s2 )</td>
<td>complex subtraction Complex ← Complex × Complex</td>
</tr>
<tr>
<td>CDIV</td>
<td>( CDIV d s1 s2 )</td>
<td>complex division Complex ← Complex × Complex</td>
</tr>
<tr>
<td>CMUL</td>
<td>( CMUL d s1 s2 )</td>
<td>complex multiplication Complex ← Complex × Complex</td>
</tr>
<tr>
<td>CMOV</td>
<td>( CMOV d s2 )</td>
<td>complex move Complex ← Complex</td>
</tr>
<tr>
<td>CNEG</td>
<td>( CNEG d s2 )</td>
<td>complex negation Complex ← Complex</td>
</tr>
<tr>
<td>CINV</td>
<td>( CINV d s2 )</td>
<td>complex reciprocal Complex ← Complex</td>
</tr>
<tr>
<td>CARG</td>
<td>( CARG d s2 )</td>
<td>complex arg Float ← Complex</td>
</tr>
<tr>
<td>CONJ</td>
<td>( CONJ d s2 )</td>
<td>complex conjugate Complex ← Complex</td>
</tr>
<tr>
<td>CABS</td>
<td>( CABS d s2 )</td>
<td>complex absolute Float ← Complex</td>
</tr>
<tr>
<td>CIPOW</td>
<td>( CIPOW d s1 s2 )</td>
<td>complex integer power Complex ← Complex × Word</td>
</tr>
</tbody>
</table>

#### Table F.6: Control operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>( BR s2 )</td>
<td>branch unconditionally to s2</td>
</tr>
<tr>
<td>BRI</td>
<td>( BRI s2 )</td>
<td>branch unconditionally, indirectly to s2 (Word)</td>
</tr>
<tr>
<td>CALL</td>
<td>( CALL s2 )</td>
<td>call unconditionally s2</td>
</tr>
<tr>
<td>CALLI</td>
<td>( CALLI s2 )</td>
<td>call unconditionally, indirectly to s2 (Word)</td>
</tr>
<tr>
<td>NOP</td>
<td>( NOP )</td>
<td>do nothing</td>
</tr>
<tr>
<td>SEQ</td>
<td>( SEQ s1 s2 )</td>
<td>sequentialise s1 ; s2</td>
</tr>
<tr>
<td>PAR</td>
<td>( PAR s1 s2 )</td>
<td>potentially parallel s1</td>
</tr>
<tr>
<td>IF</td>
<td>( IF s0 s1 s2 )</td>
<td>if (s0) s1 else s2</td>
</tr>
<tr>
<td>EVAL</td>
<td>( EVAL d s1 s2 )</td>
<td>executes s1 and returns a result from s2</td>
</tr>
<tr>
<td>VAL</td>
<td>( VAL d s2 )</td>
<td>returns the result of s2</td>
</tr>
<tr>
<td>PFUN</td>
<td>( PFUN d s2 )</td>
<td>returns a pointer to function (node) s2</td>
</tr>
<tr>
<td>RRW</td>
<td>( RRW d s1 s2 )</td>
<td>multiway data-dependency (see section 5.2.4)</td>
</tr>
<tr>
<td>WWR</td>
<td>( WWR d s1 s2 )</td>
<td>multiway data-dependency (see section 5.2.4)</td>
</tr>
<tr>
<td>SPAWN</td>
<td>( SPAWN s1 s2 )</td>
<td>spawns s1, s2 times</td>
</tr>
<tr>
<td>LOOP</td>
<td>( LOOP s0 s1 s2 )</td>
<td>call loop s0, s1 times, with stride registers given in the (common multiplier) CMB tree s2</td>
</tr>
<tr>
<td>CMB</td>
<td>( CMB s1 s2 )</td>
<td>Common multiplier tree constructor, leaves of tree are strides</td>
</tr>
<tr>
<td>INDEX</td>
<td>( INDEX d s2 )</td>
<td>Takes the value of a stride register within the context of a loop</td>
</tr>
<tr>
<td>Function</td>
<td>Explanation</td>
<td>Type compatibility</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>EVEN</td>
<td>( \text{d} = \text{even}(\text{s2}) )</td>
<td>integer even test</td>
</tr>
<tr>
<td>ODD</td>
<td>( \text{d} = \text{odd}(\text{s2}) )</td>
<td>integer odd test</td>
</tr>
<tr>
<td>NOT</td>
<td>( \text{d} = \neg \text{s2} )</td>
<td>integer (logical) not</td>
</tr>
<tr>
<td>EQ</td>
<td>( \text{d} = (\text{s1} == \text{s2}) )</td>
<td>integer equals test</td>
</tr>
<tr>
<td>NE</td>
<td>( \text{d} = (\text{s1} != \text{s2}) )</td>
<td>integer not equals test</td>
</tr>
<tr>
<td>LT</td>
<td>( \text{d} = (\text{s1} &lt; \text{s2}) )</td>
<td>integer less than test</td>
</tr>
<tr>
<td>LE</td>
<td>( \text{d} = (\text{s1} \leq \text{s2}) )</td>
<td>integer less than or equals test</td>
</tr>
<tr>
<td>GT</td>
<td>( \text{d} = (\text{s1} &gt; \text{s2}) )</td>
<td>integer greater than test</td>
</tr>
<tr>
<td>GE</td>
<td>( \text{d} = (\text{s1} \geq \text{s2}) )</td>
<td>integer greater than or equals test</td>
</tr>
<tr>
<td>LTU</td>
<td>( \text{d} = (\text{s1} &lt; \text{s2}) )</td>
<td>integer less than (unsigned) test</td>
</tr>
<tr>
<td>LEU</td>
<td>( \text{d} = (\text{s1} \leq \text{s2}) )</td>
<td>integer less than or equals (unsigned) test</td>
</tr>
<tr>
<td>GTU</td>
<td>( \text{d} = (\text{s1} &gt; \text{s2}) )</td>
<td>integer greater than (unsigned) test</td>
</tr>
<tr>
<td>GEU</td>
<td>( \text{d} = (\text{s1} \geq \text{s2}) )</td>
<td>integer greater than or equals (unsigned) test</td>
</tr>
<tr>
<td>FEQ</td>
<td>( \text{d} = (\text{s1} == \text{s2}) )</td>
<td>floating point equals test</td>
</tr>
<tr>
<td>FNE</td>
<td>( \text{d} = (\text{s1} != \text{s2}) )</td>
<td>floating point not equals test</td>
</tr>
<tr>
<td>FGT</td>
<td>( \text{d} = (\text{s1} &gt; \text{s2}) )</td>
<td>floating point greater than test</td>
</tr>
<tr>
<td>FGE</td>
<td>( \text{d} = (\text{s1} \geq \text{s2}) )</td>
<td>floating point greater than or equals test</td>
</tr>
<tr>
<td>FLT</td>
<td>( \text{d} = (\text{s1} &lt; \text{s2}) )</td>
<td>floating point less than test</td>
</tr>
<tr>
<td>FLE</td>
<td>( \text{d} = (\text{s1} \leq \text{s2}) )</td>
<td>floating point less than or equals test</td>
</tr>
<tr>
<td>CEQ</td>
<td>( \text{d} = (\text{s1} == \text{s2}) )</td>
<td>complex equals test</td>
</tr>
<tr>
<td>CNE</td>
<td>( \text{d} = (\text{s1} != \text{s2}) )</td>
<td>complex not equals test</td>
</tr>
</tbody>
</table>

Table F.7: Tests
<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
<th>Type compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP</td>
<td>(COMP complex.number)</td>
<td>atomic complex value</td>
</tr>
<tr>
<td></td>
<td>d = complex.number</td>
<td>Complex ←</td>
</tr>
<tr>
<td>STR</td>
<td>(STR string.constant)</td>
<td>address of atomic string value</td>
</tr>
<tr>
<td></td>
<td>d = &amp;string.constant</td>
<td>Word ←</td>
</tr>
<tr>
<td>FLOAT</td>
<td>(FLOAT float.number)</td>
<td>floating point value</td>
</tr>
<tr>
<td></td>
<td>d = float.number</td>
<td>Float ←</td>
</tr>
<tr>
<td>INT</td>
<td>(INT integer.number)</td>
<td>integer value</td>
</tr>
<tr>
<td></td>
<td>d = integer.number</td>
<td>Word ←</td>
</tr>
<tr>
<td>BYTE</td>
<td>(BYTE byte.number)</td>
<td>byte value</td>
</tr>
<tr>
<td></td>
<td>d = byte.number</td>
<td>Byte ←</td>
</tr>
<tr>
<td>ALLOH</td>
<td>(ALLOH d s1 s2)</td>
<td>allocate memory blocks (s2 blocks of size s1)</td>
</tr>
<tr>
<td></td>
<td>d = calloc(s1,s2)</td>
<td>Word ← Word × Word ; aligned to s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allocated on the heap (see section 5.3.7)</td>
</tr>
<tr>
<td>ALLOC</td>
<td>(ALLOC d s1 s2)</td>
<td>allocate memory blocks (s2 blocks of size s1)</td>
</tr>
<tr>
<td></td>
<td>d = calloc(s1,s2)</td>
<td>Word ← Word × Word ; aligned to s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allocated on the main stack (see section 5.3.7)</td>
</tr>
<tr>
<td>ALLOS</td>
<td>(ALLOS d s1 s2)</td>
<td>allocate memory blocks (s2 blocks of size s1)</td>
</tr>
<tr>
<td></td>
<td>d = calloc(s1,s2)</td>
<td>Word ← Word × Word ; aligned to s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allocated on the temporary variable two-stack (see section 5.3.7)</td>
</tr>
<tr>
<td>FREH</td>
<td>(FREH s2)</td>
<td>free memory blocks allocated on the heap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Word</td>
</tr>
<tr>
<td>FREE</td>
<td>(FREE s2)</td>
<td>free memory blocks allocated on the main heap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Word</td>
</tr>
<tr>
<td>FRES</td>
<td>(FRES s2)</td>
<td>free memory blocks allocated on the two-stack for temporary variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Word</td>
</tr>
<tr>
<td>EXIT</td>
<td>(EXIT)</td>
<td>tidies up and abruptly terminates the program</td>
</tr>
<tr>
<td>CSIZ</td>
<td>(CSIZ)</td>
<td>global memory for call-size variable</td>
</tr>
<tr>
<td>XTMP</td>
<td>(XTMP)</td>
<td>global memory for temporary integer variable</td>
</tr>
<tr>
<td>DICP</td>
<td>(DICP)</td>
<td>global memory for dictionary entry</td>
</tr>
<tr>
<td>TUPREF</td>
<td>(TUPREF d s2)</td>
<td>dereferences label (s2)</td>
</tr>
<tr>
<td>TUPHIREF</td>
<td>(TUPHIREF d s2)</td>
<td>high-half of address (label) of s2</td>
</tr>
<tr>
<td>TUPLOREF</td>
<td>(TUPLOREF d s2)</td>
<td>low-half of address (label) of s2</td>
</tr>
<tr>
<td>RET</td>
<td>(RET d s2)</td>
<td>Word register coalescing</td>
</tr>
<tr>
<td>FRET</td>
<td>(FRET d s2)</td>
<td>Floating point register coalescing</td>
</tr>
<tr>
<td>CRET</td>
<td>(CRET d s2)</td>
<td>Complex register coalescing</td>
</tr>
<tr>
<td>RZERO</td>
<td>(RZERO d)</td>
<td>Integer register zero (has zero value)</td>
</tr>
<tr>
<td></td>
<td>d = 0</td>
<td>Word ←</td>
</tr>
<tr>
<td>FZERO</td>
<td>(FZERO d)</td>
<td>Floating register zero (has zero value)</td>
</tr>
<tr>
<td></td>
<td>d = 0.0</td>
<td>Float ←</td>
</tr>
</tbody>
</table>

Table F.8: Memory control operations
Appendix G

Targetting the i860 from T-code

NUM(n) \equiv (\text{int}, \ldots, (n))

WORD_LENGTH \equiv \text{NUM}(32)

RZERO \equiv (\text{rzero}, \ldots, 1, 1)

PAIR(d, l, h) \equiv (\text{pair}, d, l, h)

SPLIT(t) \equiv \begin{cases} \text{(spairl, X, X, t), (spairh, X, X, Z)} & \text{for single} \\ \text{(dpairl, X, t), (dpairh, X, Z)} & \text{for double} \end{cases}

COMPLEX(t) \equiv \begin{cases} \text{(pair, d, l, t), (pair, d, l, t), tx = pair} & \text{for single} \\ \text{(pair, d, l, t), (pair, d, l, t), tx = pair} & \text{for double} \end{cases}

BNBC \equiv (\text{fld}(d), \ldots, \text{LOLAB}(\text{bnbc}), \text{HILAB}(\text{bnbc}))

FTWO \equiv (\text{fld}(l), \ldots, \text{LOLAB}(\text{ftwo}), \text{HILAB}(\text{ftwo}))

DTWO \equiv (\text{fld}(d), \ldots, \text{LOLAB}(\text{dtwo}), \text{HILAB}(\text{dtwo}))

FPOINT5 \equiv (\text{fld}(l), \ldots, \text{LOLAB}(\text{fpoint5}), \text{HILAB}(\text{fpoint5}))

DPOINT5 \equiv (\text{fld}(d), \ldots, \text{LOLAB}(\text{dpoint5}), \text{HILAB}(\text{dpoint5}))

LOLAB(name) \equiv (\text{label}, \ldots, (\text{name}), \text{RZERO})

HILAB(name) \equiv (\text{orh}, \ldots, (\text{labelh}, \ldots, (\text{name})), \text{RZERO})

CO(t) \equiv \begin{cases} x = (\text{and}, \ldots, \text{NUM}(1), (\text{shr}, \ldots, \text{NUM}(2), (\text{idc}, \ldots, 1, (\text{psr}, \ldots, 1, 1)))) \\ t = (\text{eval}, \ldots, (\text{seq}, l, t, x), x) \end{cases}

\text{dirl}(d, s1, s2) \equiv (\text{eval}, d, t1, t2)

t1 = (\text{seq}, l, (\text{shr}, \ldots, (\text{subs}, \ldots, \text{WORD_LENGTH}, s1), \text{RZERO}))

t2 = (\text{shrld}, \ldots, s2, s2)

\text{dorl}(d, s1, s2) \equiv (\text{eval}, d, t1, t2)

t1 = (\text{seq}, l, (\text{shr}, \ldots, s1, \text{RZERO}))

t2 = (\text{shrld}, \ldots, s2, s2)

\text{movl}(d, l, s2) \equiv (\text{ori}, d, \text{RZERO}, s2)

\text{nopl}(d, l, l) \equiv (\text{shrl}, l, \text{RZERO}, \text{RZERO})

\text{fnopl}(d, l, l) \equiv (\text{shrl}, l, \text{RZERO}, \text{RZERO})

\text{mulsl}(d, s1, s2) \equiv (\text{fxfr}, d, l, (\text{fmul}, \ldots, (\text{fxfr}, \ldots, s1, (\text{fxfr}, \ldots, l, s2))))

\text{mulul}(d, s1, s2) \equiv (\text{fxfr}, d, l, (\text{fmul}, \ldots, (\text{fxfr}, \ldots, s1, (\text{fxfr}, \ldots, l, s2))))

\text{fdivl}(d, s1, s2) \equiv t \ (\text{by Newton Raphson approximation})

\text{t6} = (\text{frecp}, \ldots, 1, s2)

\text{t8} = (\text{fsub}, \ldots, s2, s6)

\text{t10} = \text{DTWO}

\text{t2} = (\text{fsub}, \ldots, s2, s6)

\text{t6} = (\text{fsub}, \ldots, s2, s6)

\text{t8} = (\text{fsub}, \ldots, s2, s6)

\text{t6} = (\text{fsub}, \ldots, s2, s6)

\text{t6} = (\text{fsub}, \ldots, s2, s6)
APPENDIX G. TARGETTING THE I860 FROM T-CODE

\[ t_{87} = (\text{fmul}, \ldots, t_{65}, s_1) \]
\[ t = (\text{fmul}, d, t_{87}, t_{65}) \]

\[ t \] (by Newton Raphson approximation)
\[ t_{3} = (\text{frcp}, \ldots, \perp, s_2) \]
\[ t_{4} = (\text{fmul}, \ldots, t_{2}, s_2) \]
\[ t_{5} = \text{FTWO} \]
\[ t_{2} = (\text{fsub}, \ldots, t_{5}, t_{4}) \]
\[ t_{3} = (\text{fmul}, \ldots, t_{3}, t_{2}) \]
\[ t_{4} = (\text{fsub}, \ldots, t_{1}, t_{2}) \]
\[ t_{5} = (\text{fml}, \ldots, s_1, t_{2}) \]
\[ t = (\text{fml}, d, t_{4}, t_{5}) \]

\[ \{\text{f sqr}, d, \perp, s_2\} \]
\[ \{\text{double}\} \]
\[ t \] (by Newton Raphson approximation)
\[ t_{3} = (\text{f fgr}, \ldots, \perp, s_2) \]
\[ t_{4} = (\text{fml}, \ldots, t_{3}, s_2) \]
\[ t_{5} = \text{gPOINT5} \]
\[ t_{2} = (\text{fml}, \ldots, t_{5}, \text{f fgr}, \ldots, s_2, t_{4}) \]
\[ t_{3} = (\text{fml}, \ldots, t_{4}, \text{f fgr}, \ldots, s_2, t_{5}) \]
\[ t_{4} = (\text{fml}, \ldots, t_{3}, \text{f fgr}, \ldots, s_2, t_{4}) \]
\[ t = (\text{fml}, d, t_{3}, (\text{f fgr}, \ldots, t_{4}, (\text{f fgr}, \ldots, s_2, t_{4}))) \]

\[ \{\text{f pow}, d, s_1, s_2\} \]
\[ \{\text{single}\} \]
\[ t \] (by Newton Raphson approximation)
\[ t_{3} = (\text{f fgr}, \ldots, \perp, s_2) \]
\[ t_{4} = (\text{fml}, \ldots, t_{3}, s_2) \]
\[ t_{5} = \text{f FPOIN T5} \]
\[ t_{2} = (\text{fml}, \ldots, t_{5}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]
\[ t = (\text{fml}, d, t_{3}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]

\[ \{\text{f ipow}, d, s_1, s_2\} \]
\[ \{\text{single}\} \]
\[ t \] (by Newton Raphson approximation)
\[ t_{3} = (\text{f fgr}, \ldots, \perp, s_2) \]
\[ t_{4} = (\text{fml}, \ldots, t_{3}, s_2) \]
\[ t_{5} = \text{FPOIN T5} \]
\[ t_{2} = (\text{fml}, \ldots, t_{5}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]
\[ t = (\text{fml}, d, t_{3}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]

\[ \{\text{fipow}, d, s_1, s_2\} \]
\[ \{\text{double}\} \]
\[ t \] (by Newton Raphson approximation)
\[ t_{3} = (\text{f fgr}, \ldots, \perp, s_2) \]
\[ t_{4} = (\text{fml}, \ldots, t_{3}, s_2) \]
\[ t_{5} = \text{FPOIN T5} \]
\[ t_{2} = (\text{fml}, \ldots, t_{5}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]
\[ t = (\text{fml}, d, t_{3}, (\text{fadd}, \ldots, t_{4}, (\text{f div}, \ldots, s_1, t_{2}))) \]

\[ \{\text{cadd}, d, s_1, s_2\} \]
\[ \{\text{double}\} \]
\[ t \]
\[ \{\text{c m l}, d, s_1, s_2\} \]
\[ \{\text{fconv}, d, \perp, s_2\} \]
\[ \{\text{iconv}, d, (\text{f conv}, \ldots, \perp, s_1), (\text{f conv}, \ldots, \perp, s_2)) \]
\[ \{\text{c mov}, d, \perp, s_2\} \]
\[ \{\text{iconv}, d, (\text{f conv}, \ldots, \perp, s_1), (\text{f conv}, \ldots, \perp, s_2)) \]
\[ \{\text{pair}, d, \perp, s_2\} \]
\[ \{\text{f mov}, d, \perp, s_2\} \]
\[ \{\text{iconv}, d, (\text{f conv}, \ldots, \perp, s_1), (\text{f conv}, \ldots, \perp, s_2)) \]
\[(\text{ne}, \bot, s_1, s_2) \quad \text{not} \quad \bot, \bot, (\text{eq}, \bot, s_1, s_2)\]
\[(\text{lt}, \bot, s_1, s_2) \quad \text{CC}((\text{subs}, \bot, s_1, s_2))\]
\[(\text{le}, \bot, s_1, s_2) \quad \text{not} \quad \bot, \bot, (\text{gt}, \bot, s_1, s_2)\]
\[(\text{gt}, \bot, s_1, s_2) \quad \text{CC}((\text{subs}, \bot, s_2, s_1))\]
\[(\text{ge}, \bot, s_1, s_2) \quad \text{not} \quad \bot, \bot, (\text{lt}, \bot, s_1, s_2)\]
\[(\text{ltu}, \bot, s_1, s_2) \quad \text{CC}((\text{subs}, \bot, s_1, s_2))\]
\[(\text{leu}, \bot, s_1, s_2) \quad \text{not} \quad \bot, \bot, (\text{gt}, \bot, s_1, s_2)\]
\[(\text{gtu}, \bot, s_1, s_2) \quad \text{CC}((\text{subs}, \bot, s_2, s_1))\]
\[(\text{geu}, \bot, s_1, s_2) \quad \text{not} \quad \bot, \bot, (\text{lt}, \bot, s_1, s_2)\]
\[(\text{neg}, d, \bot, s_2) \quad \text{ccneg}, d, \bot, s_2\]
\[(\text{finv}, d, \bot, s_2) \quad \text{cinv}, d, \bot, s_2\]
\[(\text{frac}, d, \bot, s_2) \quad \text{frac}, d, \bot, s_2\]
\[(\text{conj}, d, \bot, s_2) \quad \text{conj}, d, \bot, s_2\]
\[(\text{abs}, d, \bot, s_2) \quad \text{abs}, d, \bot, s_2\]
\[(\text{fabs}, d, \bot, s_2) \quad \text{fabs}, d, \bot, s_2\]
\[(\text{cabs}, d, \bot, s_2) \quad \text{cabs}, d, \bot, s_2\]
\[(\text{even}, d, \bot, s_2) \quad \text{even}, d, \bot, s_2\]
\[(\text{odd}, d, \bot, s_2) \quad \text{odd}, d, \bot, s_2\]
\[(\text{not}, d, \bot, s_2) \quad \text{not}, d, \bot, s_2\]
\[(\text{xre}, \text{xim}), = \text{COMPLEX}(s_2)\]
\[t = \text{PAIR}(d, (\text{fneg}, \bot, \text{xre}), (\text{fneg}, \bot, \text{xim}))\]
\[(\text{finv}, d, \bot, s_2) \quad \rho_{\text{double}}\]
\[t\]
\[t^0 = (\text{frcp}, \ldots, \bot, s_2)\]
\[t^1 = (\text{fmul}, \ldots, s_2, t^2)\]
\[t^0 = \text{DTWO}\]
\[t^1 = (\text{fmul}, \ldots, t^2, t^2)\]
\[t^1 = (\text{fsub}, \ldots, t^0, t^2)\]
\[t^0 = \text{FMUL}\]
\[t^2 = (\text{fmul}, \ldots, t^2, t^2)\]
\[t^2 = (\text{fsub}, \ldots, t^0, t^2)\]
\[t^2 = (\text{fmul}, \ldots, t^2, t^2)\]
\[t^2 = (\text{fsub}, \ldots, t^0, t^2)\]
\[t = (\text{fmul}, \ldots, t^8, t^8)\]
\[t^0 = (\text{frcp}, \ldots, \bot, s_2)\]
\[t^2 = (\text{fmul}, \ldots, s_2, t_2)\]
\[t^0 = \text{FTWO}\]
\[t^1 = (\text{fsub}, \ldots, t^0, t_2)\]
\[t^1 = (\text{fmul}, \ldots, t^2, t^2)\]
\[t^1 = (\text{fsub}, \ldots, t^0, t_2)\]
\[t = (\text{fmul}, \ldots, t^8, t^8)\]
\[t = \text{PAIR}(d, (\text{fneg}, \bot, \text{xre}), (\text{fneg}, \bot, \text{xim}))\]
\[(\text{cinv}, d, \bot, s_2) \quad \rho_{\text{single}}\]
\[t\]
\[(x^r, x^i) = \text{COMPLEX}(s_2)\]
\[r = (\text{fadd}, \ldots, (\text{fmul}, \ldots, x^r, x^r), (\text{fmul}, \ldots, x^i, x^i))\]
\[t = \text{PAIR}(d, (\text{fdiv}, \ldots, x^r, r), (\text{fdiv}, \ldots, x^i, r))\]
\[t = \text{PAIR}(d, x^r, (\text{fneg}, \bot, x^i))\]
\[(\text{frac}, d, \bot, s_2) \quad \text{frac}, d, \bot, s_2\]
\[(\text{conj}, d, \bot, s_2) \quad \text{conj}, d, \bot, s_2\]
\[(\text{abs}, d, \bot, s_2) \quad \text{abs}, d, \bot, s_2\]
\[(\text{fabs}, d, \bot, s_2) \quad \text{fabs}, d, \bot, s_2\]
\[(\text{cabs}, d, \bot, s_2) \quad \text{cabs}, d, \bot, s_2\]
\[(\text{even}, d, \bot, s_2) \quad \text{even}, d, \bot, s_2\]
\[(\text{odd}, d, \bot, s_2) \quad \text{odd}, d, \bot, s_2\]
\[(\text{not}, d, \bot, s_2) \quad \text{not}, d, \bot, s_2\]
\[(\text{xre}, \text{xim}), = \text{COMPLEX}(s_2)\]
\[t = \text{PAIR}(d, (\text{fneg}, \bot, \text{xre}), (\text{fneg}, \bot, \text{xim}))\]
APPENDIX G. TARGETING THE I860 FROM T-CODE

\[(\text{bd}, d, \perp, s2)\]  
\[= (\text{subs}, d, \text{NUM}(1), s2)\]

\[(\text{max}, d, s1, s2)\]  
\[t^1 = (\text{if}, (\text{ge}, \perp, s1, s2), s1, s2)\]
\[t = (\text{eval}, d, t^1, (\text{multi}, \ldots, s1, s2))\]

\[(\text{fmax}, d, s1, s2)\]  
\[t^1 = (\text{if}, (\text{fge}, \perp, s1, s2), s1, s2)\]
\[t = (\text{eval}, d, t^1, (\text{multi}, \ldots, s1, s2))\]

\[(\text{min}, d, s1, s2)\]  
\[t^1 = (\text{if}, (\text{ge}, \perp, s1, s2), s2, s1)\]
\[t = (\text{eval}, d, t^1, (\text{multi}, \ldots, s1, s2))\]

\[(\text{fmin}, d, s1, s2)\]  
\[t^1 = (\text{if}, (\text{fge}, \perp, s1, s2), s2, s1)\]
\[t = (\text{eval}, d, t^1, (\text{multi}, \ldots, s1, s2))\]

\[(\text{ceq}, \perp, s1, s2)\]  
\[t^1 = \text{COMPLEX}(s1)\]
\[t = \text{COMPLEX}(s2)\]

\[(\text{cne}, \perp, s1, s2)\]  
\[t^1 = (\text{and}, \perp, \text{feq}, x^1, x^2, \text{fne}, x^1, x^2, \text{fne}, x^2, x^1)\]
\[t = (\text{or}, \perp, \text{fne}, x^1, x^2, \text{fne}, x^2, x^1)\]

\[(\text{blt}, s0, s1, s2)\]  
\[(\text{bst}, s0, s1, s2)\]  
\[(\text{wrd}, d, \perp, s2)\]  
\[(\text{btof}, d, \perp, s2)\]  
\[(\text{btoc}, d, \perp, s2)\]  
\[(\text{Hoc}, d, \perp, s2)\]  
\[(\text{cconv}, d, \perp, s2)\]  
\[(\text{cin}, d, \perp, s2)\]  
\[(\text{ere}, d, \perp, s2)\]  
\[(\text{byt}, d, \perp, s2)\]  
\[(\text{iconv}, d, \perp, s2)\]  
\[(\text{fconv}, d, \perp, s2)\]  
\[(\text{tupref}, d, \perp, s2)\]  
\[(\text{tuploref}, d, \perp, s2)\]  
\[(\text{tuphiref}, d, \perp, s2)\]
\[(i, s_0, s_1, s_2) \mapsto (\text{seg, } \bot, (\text{seg, } \bot, (\text{seg, } \bot, (\text{seg, } \bot, s_0, (\lambda, \bot, \bot, s_2'))),
(\text{seg, } \bot, s_1, (\text{br, } \bot, \bot, s_3))), s_2'), s_3)\]

\[s_2' = (\text{lab, } \bot, \bot, s_2)\]

\[s_3 = (\text{lab, } \bot, \bot, \bot)\]

\[\lambda = \text{bnc or } \lambda = \text{bc}, \text{ according to the direction of the flag produced by } s_1 \text{ (see page 175).}\]
Bibliography


A B Bolychevsky, V B Muchnick, and A V Shafarenko. Functional representation of a data-concurrent program. Internal report, Department of Electronic and Electrical Eng., Surrey University, Guildford GU2 5XH, 1992. This is a complete definition of f-code.


BIBLIOGRAPHY


BIBLIOGRAPHY


