Types for DSP Assembler Programs

Ken Friis Larsen
ken@friislarsen.net

Informatics and Mathematical Modeling
Computer Science and Engineering Section
Technical University of Denmark
November 2003

Abstract

In this dissertation I present my thesis:

> *A high-level type system is a good aid for developing signal processing programs in handwritten Digital Signal Processor (DSP) assembler code.*

The problem behind the thesis is that it if often necessary to programming software for embedded systems in assembler language. However, programming in assembler often causes numerous problems.

To test the thesis I define a model assembler language called Featherweight DSP which captures some of the essential features of a real custom DSP used in the industrial partner's digital hearing aids. I present a baseline type system which is the type system of DTAL adapted to Featherweight DSP. I then explain two classes of programs that uncovers some shortcomings of the baseline type systesm. The classes of problematic programs are exemplified by a procedure that initialises an array for reuse, and a procedure that computes point-wise vector multiplication. The latter uses a common idiom of prefetching memory resulting in out-of-bounds reading from memory. I present two extensions to the baseline type system: The first extension is a simple modification of some type rules to allow out-of-bounds reading from memory. The second extension is based on two major modifications of the baseline type system:

- Abandoning the type-invariance principle of memory locations and using a variation of alias types instead.
- Introducing aggregate types, making it possible to have different views of a block of memory, thus enabling type checking of programs that directly manage and reuse memory.

I show that both the baseline type system and the extended type system can be used to give type annotations to handwritten DSP assembler code, and that these annotations precisely and succinctly describe the requirements of a procedure. I implement a proof-of-concept type checker for both the baseline type system and the extensions. I get good performance results on a small benchmark suite of programs representative of handwritten DSP assembler code. These empirical results are encouraging and strongly suggest that it is possible to build a robust implementation of the type checker which is fast enough to be called every time the compiler is called, and thus can be an integrated part of the development process.

Preface

I started my PhD project at the Technical University of Denmark (DTU) in 1999 with professor Jørgen Staunstrup (DTU) and Professor Peter Sestoft (KVL) as supervisors. The project has been carried out within The Thomas B. Thrige Center for Microinstruments (CfM) DTU, and it has been financed in equal parts from three sources: the Danish Research Academy, the project "Resource-Constrained Embedded Systems" (RCES), and the CfM. Jørgen Staunstrup soon left DTU, and Jens Sparsø took over his role as head of CfM and my (now formal) supervisor. Early in the project the IT University of Copenhagen (ITU) was established and several faculty members responsible for the RCES project left DTU and took positions at the IT University of Copenhagen (ITU). I followed, and I have thus spent most of my time at the ITU working under the guidance of my co-supervisor Professor Peter Sestoft. In the period 1999–2000 I also had an office at the Department of Mathematics and Physics at the Royal Veterinary and Agricultural University, Denmark (KVL). In the period September 2000 to July 2001 I visited Computer Laboratory at University of Cambridge (CL) and Microsoft Research Cambridge (MSR) with Professor Mike Gordon (CL) as my host and Nick Benton (MSR) as my academic supervisor.

During the project I have been in close contact with a major Danish hearing aid company to ensure two things: that I did not just look at toy programs or tried to solve perceived problems. This company is denoted as *the industrial partner* throughout out the dissertation. The name of the company and the custom DSP described in Chapter 2 was known to the evaluation board.

Acknowledgements

Thanks ...

To Peter Sestoft my supervisor, mentor, and friend. Most of the ideas I present in this dissertation have been made in cooperation with or formed under influence of Peter. I owe a big debt of gratitude to Peter for his tremendous support.

To Jens Sparsø my other supervisor. Jens has untiringly handled many administrative complications.

To The engineers at the industrial partner in particular Brian Dam Pedersen, René Mortensen, and Jens Henrik Ovesen.

To Nick Benton and Mike Gordon who let me visit them in Cambridge. Nick and Mike have broaden my horizon and understanding of Computer Science.

To Fritz Henglein for arranging that I could have an office at University of Copenhagen. The greater moiety of this dissertation has been written in that office.

To Henrik Reif Andersen, who arranged a one year employment as Research Assistant with teaching obligations. Without this employment it would not have been financially possible for me to finish this project.

To Claudio Russo for debugging my my Engrish and being a good friend.

To Jesper Blak Møller for tuning my prose, being a dear friend, and for explaining many details about symbolic model checking to me.

To Michael Norrish for answering numerous questions about C and higher-order logic.

To Henning Niss for helping with some rule engineering at a most critical time.

To Joe Hurd, Martin Elsman, Jakob Lichtenberg, Konrad Slind, and Daryl Stewart for being superb office-mates.

To My parents for their love and support.

To Kamille for being the best thing that has happened in my life.

To Maria, my wife, for her unfailing love and support. Maria has repeatedly traded fractions of her own sanity to keep me somewhat within the definition of sane.

Contents

1 Types and DSP Assembler Language **1**
- 1.1 My Thesis . . . 1
- 1.2 A Bird's Eye View of the Project . . . 4
- 1.3 What This Dissertation is *not* About . . . 6
- 1.4 Inspirational Work . . . 7
- 1.5 Notation . . . 13
- 1.6 Outline of Dissertation . . . 13

2 Featherweight DSP **15**
- 2.1 The Custom DSP Architecture . . . 15
- 2.2 Characteristics of DSP Programs . . . 18
- 2.3 The Essence of the Custom DSP . . . 22
- 2.4 Summary . . . 31

3 Type System for Featherweight DSP **33**
- 3.1 Overview of the Type System . . . 33
- 3.2 Baseline Type System . . . 44
- 3.3 Properties of the Baseline Type System . . . 51
- 3.4 Shortcomings of the Baseline Type System . . . 54
- 3.5 Extension 1: Out of Bounds Memory Reads . . . 57
- 3.6 Extension 2: Pointer Arithmetics and Aggregate Types . . . 58
- 3.7 Summary . . . 67

4 Examples **69**
- 4.1 Worked Examples . . . 69
- 4.2 Limitations of the type system . . . 83
- 4.3 Comparison to Real Custom DSP Programs . . . 86
- 4.4 Summary . . . 87

5 Implementation **89**
- 5.1 Overview of the Checker . . . 89
- 5.2 Out of bounds rules . . . 94
- 5.3 Pointer Types and Aggregate Types . . . 95
- 5.4 Checking Presburger Formulae . . . 100

5.5 Benchmarks 102
5.6 Summary 103

6 Future Work and Related Work 105
6.1 Future Work 105
6.2 Related Work 116

7 Conclusion 121
7.1 Summary 121
7.2 Contributions 122

A Complete Example Code Listings 125
A.1 Fill an array with zeros 125
A.2 Pointwise Vector Multiplication with Prefetch 126
A.3 Matrix Multiplication 127
A.4 Sum of Imaginary Parts 132
A.5 Sum Over Complex Numbers 134

Bibliography 135

List of Figures

1.1 Point-wise vector multiplication in TAL and in C. 8
1.2 Point-wise vector multiplication in DTAL 10
1.3 Examples of store and pointer types. 12

2.1 Custom DSP architectural overview. 16
2.2 Pointwise vector multiplication in custom DSP assembler code and in C. 18
2.3 Graphical illustration of a pipeline that consists of four filters f_1, f_2, f_3, and f_4 20
2.4 Statistics for applications 21
2.5 Statistics for ROM primitives 22
2.6 Syntax for Featherweight DSP. 23
2.7 Pointwise vector multiplication in Featherweight DSP. 25
2.8 Syntax of Featherweight DSP machine configurations. 27
2.9 Operational Semantics of Featherweight DSP, small instructions. . 29
2.10 Operational Semantics of Featherweight DSP, instructions. 30

3.1 Type syntax for Featherweight DSP. 34
3.2 Overview of judgements for the baseline type system. 37
3.3 Well-formed index expressions, propositions, types, index contexts, and register files. 38
3.4 Substitutions 40
3.5 Type equality $\Delta; \phi \models \tau_1 \equiv \tau_2$. 41
3.6 Subtype relation $\Delta; \phi \models \tau_1 <: \tau_2$. 42
3.7 Typing of values and arithmetic expressions. 44
3.8 Type rules for small instructions. 46
3.9 Type rules for instructions. 47
3.10 Diagram for explaining the (**do**) rule. 48
3.11 Static semantics, instruction sequences 50
3.12 Static semantics, programs 51
3.13 Static semantics, dynamic locations 53
3.14 Initialisation of array. 55
3.15 Pointwise vector multiplication with prefetch. 56
3.16 Rule for out of bounds memory reads and refined rule for `do`-loops. 58

3.17 Type syntax for Featherweight DSP extended with locations and aggregate types. 59
3.18 Equality for pointer and aggregate types 61
3.19 Well-formed pointer types and aggregate types 62
3.20 Subtyping for pointer types, aggregate types, state types, and store types 63
3.21 Type rules for aliasing and pointer arithmetic 65
3.22 Modified typing rules for programs and memory values 66

4.1 Pointwise vector multiplication with type annotations. 70
4.2 Part of the derivation for $\Delta; \phi'_3 \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]$, just for the register `i0` 72
4.3 Initialisation of array with type annotations. 73
4.4 Pointwise vector multiplication with prefetch with type annotations. 76
4.5 Matrix multiplication. Part 1 79
4.6 Swapping the contents of two registers in a loop to illustrate the generality of the (**do**) rule. 82
4.7 Different representations of matrices 83
4.8 Type annotations for `multi_swap` using `choose`-types. 86

5.1 Extract of the implementation of the type checker in pseudo-ML . . 91
5.2 Part of the translation of a subtype check into a Presburger formula. 93
5.3 The six general cases for matching two aggregate types, each with three segments. 96
5.4 The Presburger formula for checking the subtype relation for two aggregate types, each with three segments. 99
5.5 Translation of a subtype check of aggregate types to a Presburger formula. 99
5.6 Benchmark numbers. 102

6.1 A type rule for reading from memory using `choose` types. 109
6.2 Type rules for position dependent types. 111
6.3 Comparison of different type system for low-level languages. 117

Chapter 1

Types and DSP Assembler Language

1.1 My Thesis

The thesis I shall argue in this dissertation is:

> *A high-level type system is a good aid for developing signal processing programs in handwritten Digital Signal Processor assembler code.*

Why should anybody be interested in handwritten assembler code? The last forty years has seen substantial developments of high-level languages to address the difficulties of programming in assembler. Today most applications for desktop computers and servers are written in high-level languages.

However, for *embedded software* (that is, the software part of an embedded system) the situation is different. Here we find that assembler still dominates. This is because much of hardware used for embedded systems is custom-made and good compilers for high-level languages are therefor not readily available. Furthermore, the hardware is often so resource constrained that high-level languages are simply not usable. A digital hearing aid is an example of such a resource constrained embedded system.

In this dissertation I focus on embedded software where *signal processing* is a key component. This is relevant for digital hearing aids, mobile phones, vehicles, mp3 players, audio-video-equipment, toy, weapons, and other systems that need to process, for example, sensor readings in a time critical manner. This kind of system often contains at least one Digital Signal Processor (DSP). A DSP is a special purpose CPU designed for signal processing. DSPs have an instruction set with instructions that make it possible to implement typical signal processing algorithms efficiently and succinctly. Digital hearing aids is a good example of embedded systems for signal processing because:

- digital hearing aids are inherently extremely resource constrained;
- the software consists almost exclusively of signal processing code.

1.1.1 Resource Constrained Embedded Systems

In some sense all computer systems are resource constrained, but desktop computers and servers often have enough resources so that the constraints are not a problem. Most embedded system have much harder constraints on memory size, power and overall dimensions. However their computing requirements are by no means low. Code in embedded systems must necessarily be fast, compact, and also energy-efficient. If the code is not getting the most out of the hardware then it can be necessary to use more powerful hardware in the embedded system. More powerful hardware use more energy, can have bigger physical dimensions, or can be more expensive.

Correctness of the code running in an embedded system is also important. It can be hard to upgrade the code running in an embedded system. And often it is only the manufacturer who have the equipment and knowledge to perform such an upgrade.

1.1.2 Difficulties of Assembler Language

Let us reiterate why programming in assembler language is difficult. The main reasons are:

- *The low level of abstraction.* Assembler language does not provide syntactic constructs for making abstractions (except that most assemblers have some support for macros).
- *Allows untrapped errors.* Assembler language enforces few restrictions. It is easy to make a programming error that corrupts an important data structure and this error can go undetected for an arbitrary length of time and then cause arbitrary behaviour of the program. Using the terminology of Cardelli [5] we say that assembler language permits untrapped errors. Untrapped errors can be difficult to find using testing, because a symptom of the error may only reveal itself when a seemingly unrelated action takes place.

  (Trapped errors, on the other hand, are errors that cause the computation to stop immediately. Trapped errors are not as time consuming to find as untrapped errors, because trapped errors can usually be found using simple testing.)

These two reasons also make it hard to *maintain* programs written in assembler. Hence, assembler programmers often follow strict coding conventions, including conventions for documenting code to a specific, detailed, format.

1.1.3 High-level Languages

High-level languages are often inappriate for embedded software, because it is common for a program written in a high-level language to demand an order of

magnitude more resources than a similar program written in hand-optimised assembler code. For custom-made hardware it can be difficult or costly to develop a compiler for a high-level language.

Let us try to break down the features high-level languages provide to overcome the difficulties of assembler programming:

1. *Language constructs.* To raise the level of abstraction, high-level languages provide constructs such as procedures, functions, objects, algebraic data types, threads, pattern matching, closures, records, and arrays. These constructs are a tremendous help for programming, because the programmer is liberated from the concerns of the low-level hardware details of the platform. For most applications the overhead from using these constructs are negligible. For embedded software, however, this overhead is often to unaffordable.

2. *Runtime systems.* Most high-level languages rely on a runtime system to support the high-level language constructs. The runtime system also provides support for features such as: dynamic memory allocation, runtime type inspection, thread creation, communication with the operating system, and perhaps garbage collection.

3. *Type systems.* Many high-level languages come with more or less advanced type systems. Type systems define the static semantics of programs and allow us to reject certain classes of faulty programs at compile time. In this dissertation I am only concerned with static type systems; dynamic type systems are regarded as a runtime system feature.

Of these three classes of features it is only the first two that directly impose an overhead at runtime. Type systems, on the other hand, can impose an indirect overhead because certain clever programs will be rejected as untyppable despite being correct. Still, static type systems have desirable features such as these:

- Types provide a succinct and precise notation for *documenting interfaces* of different program components. Because types are checked by the compiler, this kind of documentation is always consistent with the code.

- Types can be used to express *invariants* in the program. If the invariants are not satisfied, the compiler will report the violation with an error message. Thus, program defects (bugs) are caught early in the development process.

- Types can help *raise the abstraction level* in two ways: (1) types give the programmer a notation to describe the model she has in mind, and (2) it is possible to write generic high-level code (using, for example, function as parameters), which is error-prone in practise unless you have some tool to keep track of whether all invariants are satisfied.

Hence, it seems like a worthwhile goal to try and leverage the advances in type systems research to improve the tool support for assembler programming.

1.1.4 In This Dissertation

Morrisett et al. [34] and Xi and Harper [56] have studied how to design type systems suitable for very low-level languages and provide results that appear readily applicable. But the work by Morrisett et al. and Xi and Harper concentrates on assembler language used as *target language*, whereas I concentrates on assembler used as *source language*. In this dissertation:

- I show how to apply the techniques developed by Morrisett et al. and Xi and Harper to handwritten assembler code for digital hearing aids.
- I present a type system for digital hearing aids assembler code, and argue that the type system is useful for documenting code and catching errors.

In this dissertation, I concentrate on the following classes of errors:

- Giving nonsensical arguments to instructions.
- Inappropriate memory access, that is, writing or reading outside the intended memory block (this is sometimes called *memory safety* violation).
- Calling conventions violations.

1.2 A Bird's Eye View of the Project

This section gives a simplified account of the refinement process that lead to the formulation of my thesis presented in the previous section.

My thesis is a specific sub-problem of a more general problem statement, posed as a research challenge by the industrial partner:

> **Goal 0:** *Make it easier to develop software for our digital hearing aids.*

This was the problem statement I started with at the beginning of my Ph.D. project. I quickly reformulated it into a thesis suited for my background:

> **Thesis 1:** *Modern programming language technology can make it easier to develop software for digital hearing aids.*

This thesis is too general. The design space for solutions is too large. What does it for example mean to "make it easier to develop software"? Should our goal be to make the development time shorter; to make the software more correct; to make the resulting software faster; or to make the source code more succinct. And what means should we use: is it a huge library of useful components we are looking for; is it a new domain specific language; or is it an integrated develop

environment that aids developers with editing tasks, revision control, interactive experimentation and simulation, test suite building, documentation, and debugging? I decided that making it easier to develop software for digital hearing aids should mean that it is possible to catch certain classes of untrapped (and trapped) errors early in the development process; and that I would use a type system to reach this goal.

Thus, we now have the thesis presented in the last section:

> **Thesis 2:** *A high-level type system is a good aid for developing signal processing programs in handwritten DSP assembler code.*

But how can we test this thesis? For a type system to be a good aid in practise, there are a number of constraints that must be satisfied:

1. It should be theoretically possible to catch the kind of errors we want to avoid in the kinds of programs we want to write;

2. it must be feasible to implement a *type checker* for the type system, that is, the type system must not be overly complicated;

3. and it must be practical to *use* the type checker, that is, the type checker must not use excessive amounts of time to check programs we want to check, and we should not be forced to write unreasonable amounts of type annotations in programs we want to check.

To test the first constraint, I developed a formal model assembler language Featherweight DSP and tried to adopt DTAL to this assembler language. That is, I tested the following thesis:

> **Thesis 3:** *DTAL can be straightforwardly adapted to Featherweight DSP, and the resulting system can be used to conduct a case study to show the usefulness of such a system.*

When I tried to adapt DTAL to Featherweight DSP I found that the resulting system could not be used to catch all the kinds of errors I wanted to prevent, as we shall see in Chapter 3. Thus, this thesis had to be rejected.

After I had rejected **Thesis 3**, I worked with the following thesis:

> **Thesis 4:** *DTAL can be adapted, with some fundamental modifications, to Featherweight DSP. The resulting system is feasible to implement, and is practically to use.*

This final thesis is what this dissertation will address and demonstrate. It also shows the validity of the more general thesis stated in Section 1.1.

During the project, I have worked with the following guidelines, which to a certain extent are orthogonal to the thesis itself:

- *Support current practise.* I wanted to show that state-of-the-art research results can be transferred to the field of handwritten DSP assembler code, and give immediate results. That is, the DSP engineers should be able to transfer their expertise and domain knowledge in writing hand-optimised assembler code. This means that a radically new programming language or development methodology is inappropriate. The proposed type systems should be able to accommodate the current style of programming.

- *No new inventions.* This might sound like a strange guideline to pursue in a Ph.D. project. But the gist of this guideline is that instead of reinventing the wheel myself (perhaps in a slightly squarish shape), I would rather take some promising research results and try to apply them to the field of handwritten DSP assembler code. This way, I hope, has resulted in some more robust results. But, as we shall see in Chapter 3, I had to abandon this guideline. Since I needed to extend the DTAL type system with some novel type construct to get a useful type system.

1.3 What This Dissertation is *not* About

In this section I enumerate some subjects which are interesting to investigate when trying to harvest advances in programming language technology to make it easier to develop software for embedded DSPs. But all of these subjects are outside the scope of this dissertation and are not discussed or considered further.

- *Code generation.* Clearly the best way to overcome the difficulties of programming in assembler is simply to stop programming in assembler, and program in a high-level language, such as C. But then we need a compiler that can generate code for our high-level language of choice. To be competitive with hand-written assembler code, the code generator must utilise features usually found in embedded DSPs, such as: clusters of multiple functional units, multiple memory banks, low power operation, special instructions.

- *Design methodology.* For embedded systems, the hardware and software are sometimes designed together. This is called *co-design*. In co-design it is important to find the correct way to divide the system into parts that are implemented in software and parts that are implemented in hardware.

- *Developing new signal processing algorithms.* An important part of making a good hearing aid, for example, is to find or develop algorithms that can transform the sound in the desired way. These algorithms should be possible to implement efficiently on a DSP platform.

1.4 Inspirational Work

In this section I briefly introduce and summarise some of the work that has provided inspiration for the work presented in this dissertation. Some of it is technically related closely to my own work, some less so. We shall return to a more technical comparison in Chapter 6.

1.4.1 Typed Assembler Language

Typed assembler language (TAL) as introduced by Morrisett et al. [35, 36, 34] is a byproduct of the desire to have types available throughout the entire compilation process, right down to assembler level. Having types available for all intermediate representations is a great debugging aid when developing a compiler, and the types can be used for directing optimisations. Thus, TAL is designed to be machine-generated rather than handwritten.

The basic idea of TAL is to take a conventional assembler language and add type annotations to the syntax. A type checker can then check that the type annotations are correct, ensuring basic safety properties.

The TAL type system is based on a variant of the Girard–Reynolds polymorphic lambda calculus, also known as System F [17, 47]. The typing facilities provided by System F and the extensions to System F make it possible to encode high-level language features such as abstract data types, closures, objects, and continuations. This expressiveness makes TAL a more generic target language than for instance the Java Virtual Machine (JVM) bytecode [30]. The JVM instructio set is tailored to Java specific language constructs such as classes and methods.

There are several different versions and presentations of TAL with (minor) variations in the type system. The most recently described version of TAL is called Stack-based TAL [34] and is based on a model assembler language. There is also an implementation of TAL for the IA32 instruction set architecture (i.e., the Intel x86) called TALx86 to show that the techniques scale from an academic toy assembler language to a real assembler language [33]. I shall just call all these different variations TAL.

Figure 1.1 shows the TAL code, and the corresponding C function, for multiplying two vectors, point by point. The most interesting part in Figure 1.1 is the type for `vecpmult`:

```
vecpmult: ('r)
          [r0: int, r1: int array, r2: int array,
           sp: [sp: int array :: 'r] :: 'r]
```

This type succinctly describes the calling convention for `vecpmult`: arguments are in the registers `r0`, `r1`, and `r2`; the return address is the top element on the stack pointed to by `sp`; the result should be on the stack upon return; and the caller saves registers. In more detail: `vecpmult` is a label (that is what the `[]`'s

```
1  vecpmult: ('r)
2            [r0: int, r1: int array, r2: int array,
3             sp: [sp: int array :: 'r] :: 'r]
4      malloc[int] r3, 0, r0
5      mov     r4, 0
6      jmp     test
7
8  loop: ('r)
9        [r0: int, r1: int array, r2: int array,
10        sp: [sp: int array :: 'r] :: 'r,
11        r3: int array, r4: int ]
12     load    r5, r1(r4)
13     load    r6, r2(r4)
14     mul     r5, r5, r6
15     store   r3(r4), r5
16     add     r4, r4, 1
17
18 test: ('r)
19       [r0: int, r1: int array, r2: int array,
20        sp: [sp: int array :: 'r] :: 'r,
21        r3: int array, r4: int ]
22     sub     r5, r4, r0
23     blt     r5, loop
24     pop     r0
25     push    r3
26     jmp     r0
```

(a) TAL version

```
1  int* vecpmult(int n, int x[], int y[]) {
2    int *res = (int *) malloc(n*sizeof(int));
3    for(int k = 0; k < n; k++)
4       res[k] = x[k] * y[k];
5    return res;
6  }
```

(b) C version

Figure 1.1: Point-wise vector multiplication in TAL and in C.

means) of some code that expects an integer in `r0` (`r0: int`), and two integer arrays in `r1` and `r2`. The stack has this form:

```
[sp: int array :: 'r] :: 'r
```

That is, it contains at least one element. The top element of the stack is an address of some code that expects the top of stack to be an integer array: An important point to note here is how parametric polymorphism, via the stack variable `'r`, is used to abstract the shape of the stack. We can see this because `'r` occurs twice in this type.

While the example shows that it is feasible and usable to have types at assembler level, the example also shows where the TAL type system falls short. For example, we are not able to express the following requirements: the arrays in `r1` and `r2` should have the same length, n, `r0` should contain n, and the array returned on the stack will also have length n. The type system for DTAL (described in the following section) allows us to express requirements of this form. Another weakness of TAL is that it oriented towards dynamic memory allocation. TAL relies on a runtime system with a garbage collector. In addition, to preserve memory safety, all load and store instructions have to perform bounds checks at runtime. These are good design decision for the original domains for which TAL is designed, that is, as compiler intermediate language and later as a secure mobile code platform. But for embedded systems these are troublesome decisions imposing too large a runtime overhead.

1.4.2 Dependently Typed Assembler Language

Xi and Harper [56] enrich the type system of TAL with a restricted form of dependent types, called *indexed types* (sometimes also called *singleton types*). The result is called *Dependently Typed Assembler Language* (DTAL), because the types have first-order dependentcy on integer relations.

Index types are introduced to allow for more fine-grained control over memory safety so they support, for example, the elimination of array bounds check. This is done by indexing the type `int` and the type constructor `array` with an integer expression (an *index expression*): `int(`e`)` and `array(`e`)`. The meaning of indexed types is that every integer expression of type `int(`e`)` must have value equal to e and all arrays of type `array(`e`)` must have e elements. Only *Presburger arithmetic* is allowed in the index expressions, that is, integer variables, integer constants, additions, and multiplication with constants. Index expressions may contain variables, bound in an *index context*. The index context also contains a Presburger formula that constraints the domain of variables. Presburger formulae allow quantifiers over integer variables, relation expressions over Presburger expressions, and the usual Boolean connectives. Presburger arithmetic is a decidable theory (see Presburger 43 or Hopcroft and Ullman 25, page 354).

```
1  vecpmult: ('r){n:nat}
2            [r0: int(n), r1: int array(n), r2: int array(n),
3             sp: [sp: int array(n) :: 'r] :: 'r]
4      malloc[int] r3, 0, r0
5      mov     r4, 0
6      jmp     test
7
8  loop: ('r){n:nat, k:nat | k < n}
9        [r0: int(n), r1: int array(n), r2: int array(n),
10        sp: [sp: int array(n) :: 'r] :: 'r,
11        r3: int array(n), r4: int(k) ]
12     load    r5, r1(r4)
13     load    r6, r2(r4)
14     mul     r5, r5, r6
15     store   r3(r4), r5
16     add     r4, r4, 1
17
18 test: ('r){n:nat, k:nat}
19       [r0: int(n), r1: int array(n), r2: int array(n),
20        sp: [sp: int array(n) :: 'r] :: 'r,
21        r3: int array(n), r4: int(k) ]
22     sub     r5, r4, r0
23     blt     r5, loop
24     pop     r0
25     push    r3
26     jmp     r0
```

Figure 1.2: Point-wise vector multiplication in DTAL

To ensure that the type system is decidable, only Presburger arithmetic is allowed in the index expressions. That is, integer variables, quantifiers over integer variables, integer constants, addition, and subtraction [43].

Figure 1.2 shows the DTAL version of point-wise vector multiplication. Compared to the TAL version in Figure 1.1(a) only the type annotations have changed. The type annotations have only been changed by adding index expressions and index contexts. These are the underlined parts in Figure 1.2.

The most interesting type annotation in Figure 1.2 is the type for the label `loop`:

```
loop: ('r){n:nat, k:nat | k < n}
      [r0: int(n), r1: int array(n), r2: int array(n),
       sp: [sp: int array(n) :: 'r] :: 'r,
       r3: int array(n), r4: int(k) ]
```

The type specifies that before control is transferred to the code at `loop` we must satisfy that `r0` contains the natural number n, the registers `r1`, `r2`, and `r3` contain integer arrays with n elements, and `r4` contains a natural number, k, that is strictly smaller than n. The DTAL types ensures that the `load` and `store` instructions are safe although they do not perform any bounds checks at runtime. Nevertheless, DTAL still relies on a runtime system with a garbage collector.

1.4.3 Alias Types

The common technique for proving type safety for a language with imperative memory operations is based upon *type-invariance of memory locations*. That is, that the type of a given memory location must not change during the evaluation of a program. When this invariant is maintained it is straightforward to prove a subject-reduction or type-preservation property [54, 23]. The drawback is that type-invariance makes it difficult to support memory reuse and initialisation in a nice manner. The type τ of a memory location ℓ cannot change, so it must initially have type τ and after each evaluation step ℓ must still have type τ.

Alias types by Smith et al. [50], Walker and Morrisett [52]; and [53, Chapter 3] are an alternative to the type-invariance principle, designed for low-level languages such as TAL. Alias types track alias information in the type system, and make it sound to have memory locations that can hold objects of different types during evaluation. Thus, alias types allow memory reuse, sharing, and initialisation.

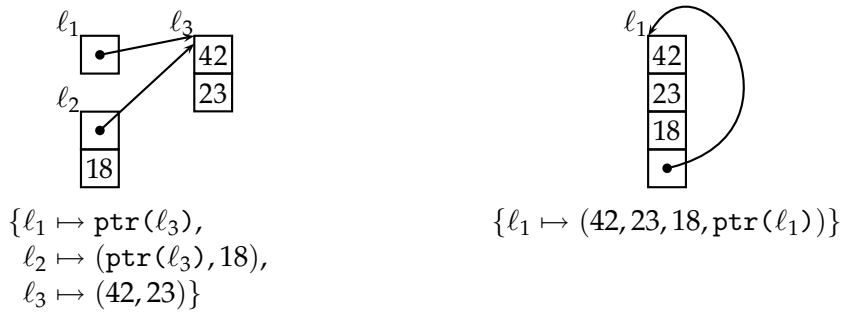
The basic ideas behind alias types are: to introduce one level of indirection to the type system by making the store visible in the types, and use singleton types to keep track of alias information by ensuring that names of memory locations are unique. That is, the basic parts of alias type are:

- A *store type* (also called an *aliasing constraint*) that is a finite mapping from locations to types.

- For a given location ℓ the *type for a pointer* to that location is `ptr(`ℓ`)`. This type is a singleton type, any pointer described by the type `ptr(`ℓ`)` is a pointer to the one location ℓ and to no other location.

Figure 1.3 show some examples of using alias types. Figure 1.3(a) shows a store with a pair at location ℓ_3, at location ℓ_1 is a pointer to location ℓ_3, and at location ℓ_2 is a pair where the first component is a pointer to location ℓ_3 and the second component is an integer. Figure 1.3(b) show a cyclic structure: a single location ℓ_1 that contains a quadruple where the last component is a pointer back to location ℓ_1.

To this basic idea, alias types add the following type-theoretic abstraction mechanisms:



(a) A shared pair (b) A cyclic structure

Figure 1.3: Examples of store and pointer types.

Location Polymorphism Often a specific piece of code does not depend on a specific location ℓ in memory. *Location polymorphism* introduce location variables ρ. Enabling code which is independent of absolute locations.

Store Polymorphism A specific procedure only operates over a portion of the store. To use that procedure in multiple contexts, the irrelevant portions of the store are abstracted away using *store polymorphism*, that is, by introducing store variables ϵ. For example, a store described by the type $\epsilon + \{\ell \mapsto \tau\}$ is a store of some unknown size and shape ϵ as well as a location ℓ containing values of type τ, where all the locations in ϵ are distinct from ℓ.

Walker and Morrisett [52] and [53, Chapter 3] also describe how tagged unions and recursive types can be handled. Alias type have been used in some versions of TAL.

1.4.4 Cyclone

Cyclone is a safe dialect of C described in [28, 22]. Cyclone shares many goals with the work presented in this dissertation—which is not surprising because both Cyclone and my work are based on TAL. Cyclone is a low-level language with a high-level type system. Cyclone is targeted at handwritten code. But, Cyclone is not targeted at embedded software and makes trade-offs which are inappropriate for resource embedded systems. The focus for Cyclone is to make it possible to build secure system-level software for desktop computers. I have not used any techniques directly from Cyclone, but Cyclone has been inspirational for the emphasis on supporting existing practise and handling existing programs.

1.4.5 Separation Logic

Separation logic and *bunched implications* by Reynolds [48], Ishtiaq and O'Hearn [26], O'Hearn and Pym [39] is an extension of Hoare-logic [24] that permits reasoning about low-level imperative programs that use shared mutable data structures.

While I have not used any particular technique from this work, separation logic has been inspirational for the way I handle locations in Chapter 2 and Chapter 3.

1.5 Notation

Finite maps are ubiquitous in the presented static and dynamic semantics. A *finite map* is a function with finite domain. If F is a finite map and $F(x) = y$ we say that x is bound to y in F. The map that (only) binds x_i to y_i for $1 \leq i \leq n$ is written $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ or $\{x_1 : y_1, \ldots, x_n : y_n\}$; the empty map (i.e., the map with domain $\emptyset$) is written $\emptyset$ or $\{\}$. We denote the of F by $\mathbf{dom}(F)$ and the range of F by $\mathbf{rng}(F)$. To extend a finite mapping F we use the syntax $F\{x \mapsto v\}$ which maps x to v even if x is already in the domain of F. We lift this so that we can compose one mapping F_1 with an other mapping F_2:

$$(F_1 + F_2)(x) \equiv \begin{cases} F_2(x) & \text{if } x \in \mathbf{dom}(F_2), \\ F_1(x) & \text{otherwise.} \end{cases}$$

1.6 Outline of Dissertation

The rest of this dissertation is organised as follows. Chapter 2 provides details about the custom DSP used in the industrial partner's hearing aids and the programming style used when programming for embedded DSPs and introduces a simple model assembler language called Featherweight DSP. In Chapter 3 I present a DTAL type system adapted for Featherweight DSP, discuss the shortcomings of this system, and I present an extended version of the type system based on alias types. Chapter 4 contains some examples. Chapter 5 gives an overview of my proof-of-concept implementation of a type checker for Featherweight DSP, and presents experimental results. In Chapter 6 I discuss how the work presented in this dissertation can be extended, compare work to related work, and discuss how this work could be used in a bigger context. Finally, Chapter 7 summarises my contributions and concludes.

Chapter 2

Featherweight DSP

As stated in Chapter 1, the focus of this dissertation is assembler programs for embedded systems where digital signal processing is a key component. That is, the embedded systems containing a *Digital Signal Processors* (DSPs).

This chapter gives an cursory overview of the assembler language for the custom DSP used in the industrial partner's hearing aids. This description is both a description of the custom DSP hardware and also a description of typical programs for this DSP. I present some statistics for the code found in the industrial partner's hearing aids.

Finally I present a formal model assembler language, called Featherweight DSP, that captures the important features of the full assembler language for the custom DSP.

2.1 The Custom DSP Architecture

This section describes the custom DSP hardware. The intention of this section is to give an intuitive feeling of the custom DSP platform, and to give a quick survey of the various architectural features commonly found on embedded DSPs. The description of the hardware is not meant to be a reference description useful for, for example, a compiler implementor. Hence, this section does not contain a complete listing of the custom DSP instruction set. Figure 2.1 shows the custom DSP architecture.

2.1.1 Registers

The custom DSP processor has five sets of registers: accumulators (two kinds named an and bn, $n = 0, 1, 2, 3$), data registers (two kinds named xn and yn, $n = 0, 1, 2, 3$), index registers (one kind named in, $n = 0, \ldots, 10$), modulo–offset registers (two kinds named mn and nn, $n = 0, \ldots, 10$), and some program control registers (described in Section 2.1.5).

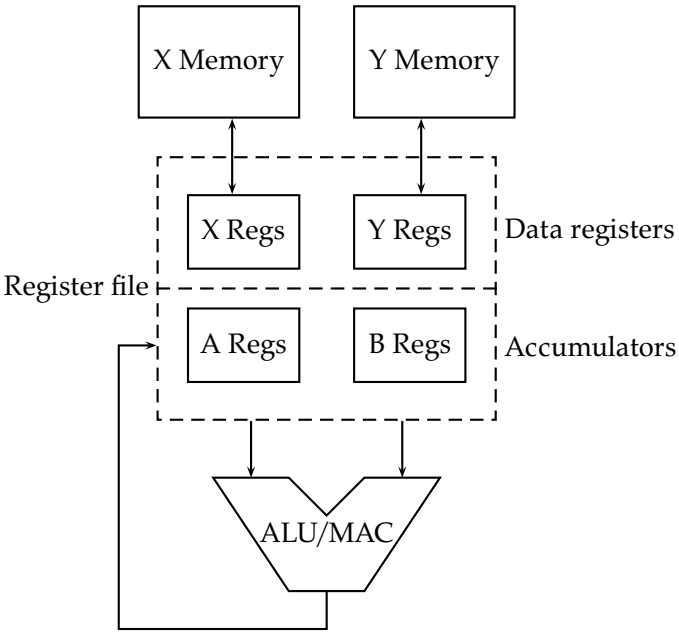


Figure 2.1: Custom DSP architectural overview.

2.1.2 Instruction-level Parallelism

The custom DSP is a *static super-scalar architecture* (sometimes called a *very long instruction word (VLIW) architecture*). This means that some instructions can be composed and executed in parallel, in effect forming a "super-instruction", also called a *composite instruction*. In particular, certain arithmetic operations may be performed in parallel with one or two data memory access operations.

2.1.3 Memory and Data Paths

There are three memory banks, one bank for code and two banks for data: X memory and Y memory. There are two *data paths*: one from X memory over x*n* data registers to a*n* accumulators, and another from Y memory over y*n* data registers to b*n* accumulators. These data paths determine which instructions can be executed in parallel. Data access on the two data paths can be performed in parallel.

2.1.4 Zero-overhead Looping Hardware

The custom DSP features *zero-overhead looping* hardware. This is specialised hardware to supports efficient execution of loops. That is, the custom DSP has special hardware support for simple loops, so the loops can be executed without incurring the loop-index-variable-update and conditional-branching overhead normally associated with loops implemented in software.

On the custom DSP, loops can be nested (up to a constant depth). The looping hardware is invoked by the `do` instruction, so we call these loops *do-loops*.

2.1.5 Custom DSP Specifics

This section describes some features in terminology specific to the custom DSP. The features are not unique to the custom DSP and variants can be found on other DSPs.

Program Control Unit

The program control unit consists of a *program counter* (PC), two stacks (a call stack and a stack for nested loops called the do-stack), and two control registers: the *mode register* (MR) and the *condition code register* (CCR).

The two stack pointers are contained in the MR. The MR also controls whether interrupts are enabled or disabled, and whether data should be shifted, rounded, or saturated when moved from accumulators to data registers or to memory.

The CCR is used for conditional branches and to detect whether the data in an accumulator need to be shifted (when the data are moved to data registers or memory) to minimise loss of precision. The CCR is also used to detect whether precision has been lost (limiting).

Addressing Modes

The custom DSP supports two addressing modes: *direct addressing* with an absolute address in store, *indirect addressing* where the address is in an index register. The indirect addressing mode allows the index register that contains the address to be *auto incremented*. There are three modes for the auto incrementation: *linear* where the hardware adds a constant to the address in the index register, *modulo* where the hardware increments the address in the index register with 1 modulo a constant, and *reverse binary* which is used to traverse the elements of a block of data in reverse binary order (bit reverse order).

The modulus–offset registers are used to control the auto incrementation mode. In modulo mode, only a restricted set of constants can be used as the modulus (the first fourteen powers of 2).

Peripheral Space

External units may be attached to the custom DSP core processor. These external units, and some of the internals of the custom DSP processor itself are accessed and controlled through *peripheral space*.

```
1 vecpmult:
2     do (i7), lend
3        x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
4        a0=x0*y0
5 lend:  xmem[i1] = a0; i1+=1
6     ret
```

(a) Custom DSP version

```
1 void vecpmult(int len, float x[], float y[], float result[]) {
2   int i;
3   for(i = 0; i < len; i++)
4     result[i] = x[i] * y[i];
5 }
```

(b) C version

Figure 2.2: Pointwise vector multiplication in custom DSP assembler code and in C.

2.1.6 Example code: Pointwise vector multiplication

Figure 2.2 shows custom DSP assembler code and corresponding C code for computing pointwise vector multiplication. In Figure 2.2(a), line 2 through line 5 provide an example of a `do`-loop. That is, the `do` instruction (line 2) takes two arguments: the number of loop iterations, `i7`, and the address of the last instruction, `lend`. Line 3 is an example of a composite instruction where two memory loads are executed in parallel. Each of the two loads use indirect addressing and auto increment the index registers `i0` and `i4`. Line 5 shows how a register can be stored to memory and that auto increment also works for store operations.

Compared to the C code the register `i7` corresponds to variable `len`, register `i0` corresponds to the variable `x`, register `i4` corresponds to the variable `y`, and the register `i1` corresponds to the variable `result`. The C variable `i` does not have a custom DSP counterpart, because we traverse the arrays pointed to by the registers `i0`, `i4`, and `i1` by incrementing these registers.

It is interesting to notice that the assembler syntax for the custom DSP uses infix syntax. With proper indentation of `do`-loops, the code starts to resemble high level C code.

2.2 Characteristics of DSP Programs

To design a successful type system for a domain specific assembler language it is important to exploit any domain specific patterns, and to capture frequently

used idioms.

This section presents some qualitative and quantitative characteristics of embedded DSP code. These characteristics have been identified by examination of code from [2] and from a snapshot of the code used in the industrial partner's digital hearing aids. Using the terminology and taxonomy from ordinary software we can classify the software for digital hearing aids into *operating system* and *user code*. The user code can further be classified into *application code* and *library code*. In the following, we consentrate only on the user code, because the operating system code is particular to specific features of the hardware and it is hard to extract general design patterns from this code. The only thing to say about the operating system is that it takes care of interaction with the hardware. Part of this is the interaction with the user of the hearing aid. That is, the operating system monitors the buttons and dials on the hearing aids and takes care of running the application(s) on the hearing aid.

2.2.1 Current Development Practice

The typical development process for DSP software is:

1. Experiment and design signal processing algorithms in a high-level language, often Matlab.
2. Translate (by hand) the high-level design to C and convert from floating point arithmetic to fixed point arithmetic. Test to ensure that the converted algorithm still has the desired properties.
3. Translate (by hand) the C code to DSP assembler. Test that the assembler code produces the correct results.

For an informal description of this development process see [31]. Step 2 and step 3 are especially time consuming and error prone.

2.2.2 Qualitative Characteristics

This section gives a brief overview of what DSP code looks like, when we are only concerned with general patterns and idioms.

No dynamic memory allocation: The code is arranged so that only statically allocated, fixed sized buffers (arrays) are used.

Array manipulation is everything: Signal processing algorithms are often expressed in terms of vector and matrix manipulation. The code is typically implemented using arrays.

Sequential traversal: With two noticeable exceptions, arrays are traversed in sequential order. The exceptions are fast Fourier transformation (FFT) [11] and cyclic buffers. Thus, DSPs often come with special address modes

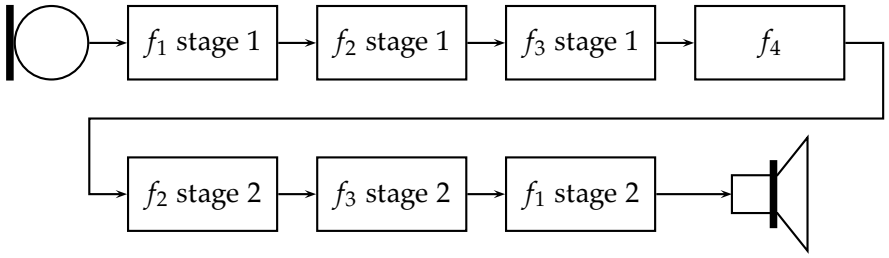


Figure 2.3: Graphical illustration of a pipeline that consists of four filters f_1, f_2, f_3, and f_4

making reverse bit indexing (used in FFT) and modulus indexing (used in cyclic buffers) look like sequential indexing to the programmer. The addressing modes of the custom DSP, described in Section 2.1.5, support these too.

No stack: DSPs often do not have a general purpose stack for transfering procedure arguments and storing local variables. Instead they have many special purpose registers and sometimes a small special purpose call-frame stack.

No recursive functions: Recursive functions are not found in DSPs code for two reasons: the hardware does not have a stack, so recursion is hard to implement; if the programmer is not careful, recursion naturally leads to unbounded use of resources.

Code is organised in small procedures: The code in both [2] and the industrial partner's digital hearing aids is organised in small procedures. Each procedure has specific and well-defined functionality, like multiplying two vectors, for instance.

No self-modifying code: I have not found any examples of self-modifying code. Furthermore the code section is usually stored in read-only-memory, and thus it is not common practice to write self-modifying code.

Pipeline organisation: There is only one application in a hearing aid, namely the filter that transforms the sound samples from the microphone before the transformed samples are played on the loudspeaker. But this one filter is typically composed of a pipeline of several simpler filters. Each filter in this pipeline can consists of several stages in the pipeline. Figure 2.3 shows a diagrammatic pipeline of four filters, three of which consist of two stages.

Number of procedure macros:	14
Number of procedures with a `do` loop:	9
Number of procedure macros with nested `do` loops:	5
Number of procedures with a local jump:	5
Number of procedures with a `call`:	5
Number of inline code macros:	7
Average size excluding comments (lines of code):	38
Average size including comments (lines of code):	59
Average percentage of the size of comments	35
Total size of all procedures excluding comments (lines of code):	530

Figure 2.4: Statistics for applications

2.2.3 Quantitative Characteristics

This section gives some more detailed and quantitative characteristics of the code used in the industrial partner's hearing aids. I describe the coding style used in the application code and in the library code and present some concrete statistics.

One of the interesting things to note is that both application code and library code are based on the procedure abstraction, but each type of code used a different style to implement this abstraction. In the following I use the word *procedure* to mean either style.

Applications: As mention in the previous section, there is only one application in a hearing aid: a filter pipeline. Thus, the application code is the code for each of the filters in this pipeline.

Each stage of a filter is implemented as a procedure, and the main style of implementing a procedure is to implement it as a macro. The justification for this is that the procedure comprising the main pipeline are just called in sequence and application procedures do not call other application procedures, so macro expansion is finite.

Figure 2.4 presents some statistics for the procedures comprising the main filter pipeline. These numbers have been collected mostly by hand. The notion of *inline macros* comes from the comments in the source code, we can think of them as helper procedures. A local jump is one that does not jump outside the procedure body.

Library code: (Also called *ROM primitives*) Each ROM primitive is implemented as a procedure, and a procedure is implemented as number of named entry points (that is, symbolic labels) and an instruction sequence ending with the return instruction, that pops the return address from the internal call-stack and jump to the return address. See Figure 2.2(a) for an example of a typical ROM primitive.

Number of procedures:	43
Number of procedures with a `do` loop:	42
Number of procedures with nested `do` loops:	4
Number of procedures with a local jump:	3
Number of procedures with a `call` or a long `jmp`:	0
Average size excluding comments (lines of code):	25
Average size including comments (lines of code):	40
Average percentage of the size of comments	37
Total size of all procedures excluding comments (lines of code):	1074
Number of calls to undefined labels (in macros):	4
Number of shared bodies:	5
Number of procedures where the first label is not a call target:	10
Number of procedures which are not a target of a `call`:	2

Figure 2.5: Statistics for ROM primitives

The code for a procedure is structured so that each procedure consists of one or more entry points to a preamble. The preamble takes care of putting the right values in the right registers and setting the addressing unit and the like in the right mode. There can be more than one entry point to the preamble, because sometimes it is more efficient to skip some of the setup code. After the preamble is the body of the procedure. The code in Figure 2.2(a) does not include a preamble.

To each procedure is associated a wrapper macro. This macro wraps the calling convention for the procedure. The reason for this macro wrapping is to make it easier to patch the preamble (simply by skipping it perhaps) and to relieve the programmer from remembering the specific calling convention for each procedure. Thus, the convention is that a procedure should never call another procedure directly, the call should always happen through the associated macro.

Figure 2.5 presents some statistics for the ROM primitives. These numbers have been collected mainly by machine. An interesting thing to note in Figure 2.5 is that there are only four procedures with nested loops. In fact, no loops are nested to more than depth 2. A shared body is one which is associated with two or more macros for different entry points to the body.

2.3 The Essence of the Custom DSP

This section presents the formal model assembly language Featherweight DSP. The language is used to capture some of the essential features of the custom DSP:

programs	P	$\equiv$	$(\ell_i,\ \ell_1 : mval_1 \quad \cdots \quad \ell_n : mval_n)$
memory values	$mval$	$\equiv$	$I \mid dval$
data values	$dval$	$\equiv$	`X:<`c_1`,...,`c_n`>`
		$\mid$	`Y:<`c_1`,...,`c_n`>`
instruction sequences	I	$\equiv$	`jmp(`v`)`
		$\mid$	`ret`
		$\mid$	`halt`
		$\mid$	$ins\ I$
instructions	ins	$\equiv$	`call(`v`)`
		$\mid$	$sins_1$`;...;`$sins_n$
		$\mid$	`do(`v`) {`B`}`
		$\mid$	`enddo`
		$\mid$	$bop\ r$`,` v
do-bodies	B	$\equiv$	$ins_1 \cdots ins_n$
small instructions	$sins$	$\equiv$	r_d `=` `xmem[`r_s`]`
		$\mid$	`xmem[`r_{md}`]` `=` r_s
		$\mid$	r_d `=` `ymem[`r_s`]`
		$\mid$	`ymem[`r_{md}`]` `=` r_s
		$\mid$	r_d `+=` $aexp$
		$\mid$	r_d `=` $aexp$
arithmetic expressions	$aexp$	$\equiv$	v
		$\mid$	r_1 `+` r_2
		$\mid$	r_1 `*` r_2
branch operators	bop	$\equiv$	`beq` | `bneq` | `bgt` | `blt` | `bgte` | `blte`
values	v	$\equiv$	$c \mid r$
constants	c	$\equiv$	$f \mid i \mid \ell$
fixed-point constants	f		
integer constants	i		
labels	ℓ		

Figure 2.6: Syntax for Featherweight DSP.

composite instructions, `do`-loops, the hardware support for procedure abstraction, and sequential traversal of arrays using pointer arithmetic. The last features is of course not specific to the custom DSP, but pointer arithmetic is usualy ignored in formal model assembly languages because it is unmanageable. Since our ultimate goal is to design a type system for the real custom DSP, it is necessary to handle some form of pointer arithmetic.

2.3.1 Syntax of Featherweight DSP

Figure 2.6 contains the syntax for Featherweight DSP, the syntax resembles the syntax of the custom DSP with only minor deviations. Like a conventional assembler language, a Featherweight DSP program consists of three parts: a set

of labelled instruction sequences, where labels are used as symbolic addresses for control transfer instructions; a set of labelled data locations, here the data can be in either X or Y memory; and start label (ℓ_i in the figure) which is where the program is started. In the syntax, I use r to represent a register operand, v to represent an operand that is either a register or an immediate word-sized value, and c to range over word-sized constants, that is, an integer i, a fixed-point number f, or a code or date label ℓ.

Small instructions

Instructions are divided into two kinds: those that can be executed in parallel in a composite instruction, and those that cannot be executed in parallel. The former kind is called small instructions, *sins*, and the latter is simply called instructions, *ins*. The syntax for small instructions should be mostly self-explanatory as it resembles the syntax of a high-level language like C. To load a value from X memory into the register r_d we write:

$$r_d \texttt{ = xmem[}r_s\texttt{]}$$

where r_s is the source register that must contain an address in X memory. And similar if we want to store the value in the register r_s to Y memory we write:

$$\texttt{ymem[}r_{md}\texttt{] = }r_s$$

where r_{md} is the memory destination register that must contain an address in Y memory.

Arithmetic operations are restricted to addition and multiplication of two registers. This is of course only a small subset of the arithmetic operations the real custom DSP provides. The real custom DSP has a multiply with pre-add:

$$r_d \texttt{ = } r_1 \texttt{ * (}r_2 \texttt{ + } r_3\texttt{)}$$

and various bit-fiddling operations like shifts, for instance. Curiously enough, the custom DSP do not have any division operation, so we obmit it too.

Composite instructions

We form composite instructions out of small instruction simply by putting semicolon between them $sins_1 ; \ldots ; sins_n$. However, we need to place certain restrictions on the small instructions in a composite instruction:

1. A register must only occur once in a destination register r_d position.
2. There must be at most one load or store from X memory.
3. There must be at most one load or store from Y memory.

```
1 vecpmult:
2     do (i7) {
3        x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
4        a0=x0*y0
5        xmem[i1] = a0; i1+=1
6     }
7     ret
```

Figure 2.7: Pointwise vector multiplication in Featherweight DSP.

We define the predicate UniqDef over composite instructions to be true if these restrictions are satisfied and false otherwise. The restrictions for Featherweight DSP are a relaxed version of the restrictions for the real custom DSP. The only property we are interested in for Featherweight DSP, is that no race conditions can occur. That is, the contents of a register or a memory location must be deterministic. Composite instructions are also used to model the auto increment feature of the load and store operations of the real custom DSP.

Loops

I have made a slightly modified syntax for `do`-loops compared to the real custom DSP. In the real custom DSP assembler language the `do` instruction takes a label denoting the last instruction of the loop body as its second argument. In Featherweight DSP the body of a `do`-loop is simply enclosed in curly braces. Figure 2.7 contains the code for pointwise vector multiplication in Featherweight DSP for comparison with the code in Figure 2.2(a) on page 18.

Contrary to what our first intuition might lead us to believe, the instruction `enddo` is *not* used to terminate a `do`-loop. The instruction `enddo` is used if we jump out of the body a `do`-loop, because the do-stack is left in an inconsistent state, and `enddo` brings the stack back into a consistent state by popping the top element of the do-stack. If we jump out of a nested loops, then `enddo` must be called as many times as the nesting is deep. Also notice that, in Featherweight DSP the instructions `jmp` and `ret` are not allowed in the body of a loop. In Featherweight DSP the instructions `do` and `enddo` are the only instructions for manipulating the do-stack. In the real custom DSP the do-stack can also be manipulated through peripheral space, but I have not found any real code that does that.

Hardware procedures

Featherweight DSP (and the real custom DSP) offers hardware support for implementing procedures using the instructions `call` and `ret`. The instruction `call` takes a code location v as its sole operand; `call` pushes the address of the

instruction following the `call` instruction onto the call-stack and then transfer control to the instruction at v. The instruction `ret` pops the top element, which is a code location, off the call-stack and jumps to this location. In Featherweight DSP the instructions `call` and `ret` are the only instructions for manipulating the call-stack. The call-stack can not be used for transferring arguments to procedures, these arguments must be transferred via registers or memory. In the real custom DSP the call-stack can also be manipulated through peripheral space, but I have not found any examples of real code that does that.

Branch instructions

In the assembler language for the real custom DSP the branch instructions has the form:

`if(`e`) jmp(`v`)`

where e is one of a finite set of expressions testing the CCR. An example of such a test is:

```
a == 0
```

which tests that the last test instruction on one of the an accumulators was zero. For example, the following two instructions tests if `a0` is zero and if so jumps to the code located at `foo`:

```
a0 & a0
if (a == 0) jmp foo
```

where `a0 & a0` is the bitwise AND test instruction (he operands of this instruction are not altered but the CCR is).

In Featherweight DSP there is no CCR, instead there are several branch instructions that takes two operands and branch to the second operand if the first operand is appropriately related to zero; otherwise execution continues with the instruction following the branch instruction. Thus, the following instruction tests whether `a0` is zero, and if, so jumps to `foo`:

```
beq a0, foo
```

I have chosen this simplification of the branch instruction, because then the type system presented in the next chapter does not have to keep track of a CCR.

Instruction sequences and control transfer instructions

An instruction sequence, I, is a list of instructions terminated by an unconditional control transfer instruction: `jmp`, `ret`, or `halt`.

machine configuration	M	$\equiv$	$(X, Y, C, \Gamma, S, D, I)$
store for X memory	X	$\equiv$	$\{\ell_1 \mapsto td_1, \ldots \ell_n \mapsto td_n\}$
store for Y memory	Y	$\equiv$	$\{\ell_1 \mapsto td_1, \ldots \ell_n \mapsto td_n\}$
store for code memory	C	$\equiv$	$\{\ell_1 \mapsto I_1, \ldots \ell_n \mapsto I_n\}$
register file	Γ	$\equiv$	$\{r_1 \mapsto d_1, \ldots r_n \mapsto d_n\}$
call-stack	S	$\equiv$	$nil \mid I :: S$
do-stack	D	$\equiv$	$nil \mid (i, B) :: D$
tuple of data values	td	$\equiv$	$(d_1, \ldots, d_n)$
data value	d	$\equiv$	$ns \mid i \mid f \mid loc$
location	loc	$\equiv$	$\langle \ell, i \rangle$

Figure 2.8: Syntax of Featherweight DSP machine configurations.

2.3.2 Dynamic Semantics for Featherweight DSP

To define the dynamic semantics for Featherweight DSP I use a standard approach and specify the semantics as an abstract rewriting machine, similar to the STAL abstract machine [34] or the SECD machine [29].

Machine Configurations

For Featherweight DSP a machine configuration M consists of seven components: a store for X memory (X), a store for Y memory (Y), a store for code memory (C), a register file (Γ), a call-stack (S), a do-stack (D), and a current instruction sequence (I). Execution is modelled by a deterministic rewriting system that transform a machine configuration M to a machine configuration M', written $M \blacktriangleright M'$.

The stores for X and Y memory are finite mappings from labels to tuples of data values, where a data value is either an integer or fixed-point constant, the special nonsense value *ns*, or a location. A location $\langle \ell, i \rangle$ is an offset label ℓ and an integer constant i, that is, a location $\langle \ell, i \rangle$ can represent the address $\ell + i$. The store for code memory is a finite mapping from labels to instruction sequences. The register file is a finite mapping from register names to data values. The call-stack is a list of instruction sequences, and the do-stack is a list of pairs where the first component of the pair is an integer and the second component is a do-body. The syntax of machine configurations is summarised in Figure 2.8 where some syntactic categories are reused from Figure 2.6 but not repeated.

In this machine model I use instruction sequences to represent code pointers. Before we specify the rewriting rules we introduce a bit of convenient notation. We use $\hat{\Gamma}(v)$ to convert an operand to a data value as follows:

$$\begin{array}{lcl} \hat{\Gamma}(r) & \equiv & \Gamma(r) \\ \hat{\Gamma}(\ell) & \equiv & \langle \ell, 0 \rangle \\ \hat{\Gamma}(c) & \equiv & c \end{array}$$

where the last clause matches integer and fixed-point constants, but not labels. For the X and Y stores we use $\hat{X}(loc)$ and $\hat{Y}(loc)$ to convert a location to a data value:

$$\hat{X}(\langle \ell, i \rangle) \equiv \begin{cases} d_{i+1} & \text{if } 0 \leq i < n, \\ ns & \text{otherwise} \end{cases}$$

where $X(\ell) = (d_1, \ldots, d_n)$.

The model of the store used in this machine model is similar to the model used for C [27]. In particular, the way stores are represented does not say anything about the physical adjacency of labels. Thus, from a given label ℓ_1 it is not possible to access the data of another label ℓ_2. For example, if we have two arrays, one with the elements 1, 2, and 3, and another with the elements 10, 20, 30, and 40. We can represent these arrays with the data declarations:

```
loc1  :  X:<1,2,3>
loc2  :  X:<10,20,30,40>
```

If we try load data from location $\langle$`loc1`$,3\rangle$, then we do not get 10, instead we get the nonsense value *ns*. Thus, if we want to be able to reach both arrays from just one location we must use the data declaration:

```
loc1  :  X:<1,2,3, 10,20,30,40>
```

(whitespace is not significant).

Rewrite Rules

To specify the semantics of Featherweight DSP we use two sets of rewrite rules, one for small instructions and another for machine configurations. We need two set of rules to handle the parallelism in composite instructions. Figure 2.9 shows the rules for small instructions and Figure 2.10 shows the rules for machine configurations. Both sets of rules are presented as inferences rules. But the rewrite system is still flat; the $\triangleright$ only occurs in a premise for the $\blacktriangleright$ relation, and the $\blacktriangleright$ relation never occurs in a premise. Hence, the system can be thought of as a machine.

The rules for small instructions works on a partial machine configuration that only consist of the stores for X and Y memory, the register file, and a single small instruction. The rules transform a partial machine configuration $(X, Y, \Gamma, sins)$ to a partial store for X memory X', a partial store for Y memory Y', and a partial register file Γ', written:

$$(X, Y, \Gamma, sins) \triangleright (X', Y', \Gamma')$$

It is important to note that the rules for small instructions only return mappings with at most one binding.

$$\frac{\hat{X}(\Gamma(r_2)) = d}{(X, Y, \Gamma, r_1 \texttt{ = xmem[}r_2\texttt{]}) \rhd (\emptyset, \emptyset, \{r_1 \mapsto d\})}$$

$$\frac{\Gamma(r_2) = d \qquad \Gamma(r_1) = \langle \ell, i \rangle \qquad X(\ell) = (d_1, \ldots, d_{i+1}, \ldots, d_n)}{(X, Y, \Gamma, \texttt{xmem[}r_1\texttt{] = }r_2) \rhd (\{\ell \mapsto (d_1, \ldots, d, \ldots, d_n)\}, \emptyset, \emptyset)}$$

$$\frac{\hat{Y}(\Gamma(r_2)) = d}{(X, Y, \Gamma, r_1 \texttt{ = ymem[}r_2\texttt{]}) \rhd (\emptyset, \emptyset, \{r_1 \mapsto d\})}$$

$$\frac{\Gamma(r_2) = d \qquad \Gamma(r_1) = \langle \ell, i \rangle \qquad Y(\ell) = (d_1, \ldots, d_{i+1}, \ldots, d_n)}{(X, Y, \Gamma, \texttt{ymem[}r_1\texttt{] = }r_2) \rhd (\emptyset, \{\ell \mapsto (d_1, \ldots, d, \ldots, d_n)\}, \emptyset)}$$

$$\frac{\Gamma(r) = f_1 \qquad [\![aexp]\!] = f_2}{(X, Y, \Gamma, r \texttt{ += } aexp) \rhd (\emptyset, \emptyset, \{r \mapsto f_1 + f_2\})}$$

$$\frac{\Gamma(r) = i_1 \qquad [\![aexp]\!] = i_2}{(X, Y, \Gamma, r \texttt{ += } aexp) \rhd (\emptyset, \emptyset, \{r \mapsto i_1 + i_2\})}$$

$$\frac{\Gamma(r) = \langle \ell, i_1 \rangle \qquad [\![aexp]\!] = i_2}{(X, Y, \Gamma, r \texttt{ += } aexp) \rhd (\emptyset, \emptyset, \{r \mapsto \langle \ell, i_1 + i_2 \rangle\})}$$

$$(X, Y, \Gamma, r \texttt{ = } aexp) \rhd (\emptyset, \emptyset, \{r \mapsto [\![aexp]\!]\})$$

Figure 2.9: Operational Semantics of Featherweight DSP, small instructions.

In the rules for small instructions we use the notation $[\![aexp]\!]$ to denote the translation of an arithmetic expression, given an implicit register file Γ:

$$\begin{aligned}
[\![v]\!] &\equiv \hat{\Gamma}(v) \\
[\![r_1 \texttt{ + } r_2]\!] &\equiv \begin{cases} f_1 + f_2 & \text{if } \Gamma(r_1) = f_1 \text{ and } \Gamma(r_2) = f_2, \\ i_1 + i_2 & \text{if } \Gamma(r_1) = i_1 \text{ and } \Gamma(r_2) = i_2, \\ \langle \ell, i_1 + i_2 \rangle & \text{if } \Gamma(r_1) = \langle \ell, i_1 \rangle \text{ and } \Gamma(r_2) = i_2, \\ \langle \ell, i_1 + i_2 \rangle & \text{if } \Gamma(r_1) = i_1 \text{ and } \Gamma(r_2) = \langle \ell, i_2 \rangle, \end{cases} \\
[\![r_1 \texttt{ * } r_2]\!] &\equiv \begin{cases} f_1 \cdot f_2 & \text{if } \Gamma(r_1) = f_1 \text{ and } \Gamma(r_2) = f_2, \\ i_1 \cdot i_2 & \text{if } \Gamma(r_1) = i_1 \text{ and } \Gamma(r_2) = i_2, \end{cases}
\end{aligned}$$

The rules for machine configurations in Figure 2.10 are directed by the current instruction sequence. The rules are straightforward and standard, except for the rule for composite instructions and the rules for `do`-loops.

To give a semantics for `do`-loops, I have introduced the special instruction `CHECK` as a purely technical device used to specify when the do-stack should be checked. At first, the device of using a special instruction might seem clumsy,

$$\frac{\hat{\Gamma}(v) = \langle \ell, 0 \rangle \qquad C(\ell) = I'}{(X, Y, C, \Gamma, S, D, \texttt{jmp(}v\texttt{)}) \blacktriangleright (X, Y, C, \Gamma, S, D, I')}$$

$$\frac{S = I' :: S'}{(X, Y, C, \Gamma, S, D, \texttt{ret}) \blacktriangleright (X, Y, C, \Gamma, S', D, I')}$$

$$\frac{\hat{\Gamma}(v) = \langle \ell, 0 \rangle \qquad C(\ell) = I''}{(X, Y, C, \Gamma, S, D, \texttt{call(}v\texttt{)} \quad I') \blacktriangleright (X, Y, C, \Gamma, I' :: S, D, I'')}$$

$$\frac{\begin{array}{c}\textsc{UniqDef}(sins_1, \ldots, sins_n) \\ (X, Y, \Gamma, sins_1) \rhd (X_1, Y_1, \Gamma_1) \quad \cdots \quad (X, Y, \Gamma, sins_n) \rhd (X_n, Y_n, \Gamma_n) \\ X' = X + X_1 + \cdots + X_n \qquad Y' = Y + Y_1 + \cdots + Y_n \qquad \Gamma' = \Gamma + \Gamma_1 + \cdots + \Gamma_n\end{array}}{(X, Y, C, \Gamma, S, D, sins_1\texttt{;} \ldots \texttt{;} sins_n \quad I') \blacktriangleright (X', Y', C, \Gamma', S, D, I')}$$

$$\frac{\hat{\Gamma}(v) = i}{(X, Y, C, \Gamma, S, D, \texttt{do(}v\texttt{)} \ \texttt{\{}B\texttt{\}} \quad I') \blacktriangleright (X, Y, C, \Gamma, S, (i, B) :: D, B \quad \texttt{CHECK} \quad I')}$$

$$\frac{D = (0, B) :: D'}{(X, Y, C, \Gamma, S, D, \texttt{CHECK} \quad I') \blacktriangleright (X, Y, C, \Gamma, S, D', I')}$$

$$\frac{D = (i, B) :: D' \qquad i \neq 0}{(X, Y, C, \Gamma, S, D, \texttt{CHECK} \quad I') \blacktriangleright (X, Y, C, \Gamma, S, (i - 1, B) :: D', B \quad \texttt{CHECK} \quad I')}$$

$$\frac{D = (i, B) :: D'}{(X, Y, C, \Gamma, S, D, \texttt{enddo} \quad I') \blacktriangleright (X, Y, C, \Gamma, S, D', I')}$$

$$\frac{\Gamma(r) \neq 0}{(X, Y, C, \Gamma, S, D, \texttt{beq } r\texttt{, } v \quad I') \blacktriangleright (X, Y, C, \Gamma, S, D, I')}$$

$$\frac{\Gamma(r) = 0 \qquad \hat{\Gamma}(v) = \langle \ell, 0 \rangle \qquad C(\ell) = I''}{(X, Y, C, \Gamma, S, D, \texttt{beq } r\texttt{, } v \quad I') \blacktriangleright (X, Y, C, \Gamma, S, D, I'')}$$

Figure 2.10: Operational Semantics of Featherweight DSP, instructions.

but without it, it is hard to give a precise semantics of the `enddo` instruction. It is not surprising that the `do` causes problems, because the construct is more high-level that the other instructions.

The rule for composite instructions uses the $\rhd$ relation for small instructions. The rule does not specify in which order the small instructions should be rewritten because it does not matter as long as the UniqDef predicate is satisfied.

For branch instructions, I only present the rules for `beq`, the rules for the other branch instructions are trivial variations.

We say that the machine is in a *terminal configuration* if the current instruction sequence is `halt`. And we say that the abstract machine is *stuck* if the machine is not in a terminal configuration but there is no rule that applies to the current configuration. The machine can become struck if we try to add a fixed-point number and an integer, try to multiply two pointers, try to use an integer as a location, or try to execute the `enddo` instruction with an empty do-stack, for instance.

2.4 Summary

In this section I have given a brief survey of the features of the custom DSP, and summarised the features particular to embedded DSPs. I have given some qualitative and quantitative characteristics of the code found in the industrial partner's digital hearing aids and similar systems.

I have also presented the formal model assembler language Featherweight DSP which is used in subsequent chapters. Finally, I define the semantics of Featherweight DSP using a set of rewrite rules specifying an abstract machine.

The contribution of this chapter is an explanation of the problems and features that are important in the domain of code for embedded DSPs.

Chapter 3

Type System for Featherweight DSP

This chapter presents a static semantics (i.e., a type system) for Featherweight DSP. I present the type system in three phases: First, I describe a baseline type system close to DTAL, but adapted to Featherweight DSP. Second, I briefly discuss some problems with the baseline type system, guided by some real-life code examples. Finally, I present two extensions to the baseline type system to overcome its limitations.

3.1 Overview of the Type System

The ultimate goal of this chapter is to define a type system for Featherweight DSP programs, in particular instruction sequences, that will enable us to catch certain classes of errors at compile time. The classes of errors we concentrate on are:

- Nonsense arguments to instructions
- Memory safety violations
- Calling convention violations

Chapter 4 shows how to use the type system to catch these kinds of errors in practise.

The type system is defined as a set of *judgements* where each judgement is defined by a set of *typing rules*. These judgements are described in the following sections. In this section I introduce the basic structure (that is, the syntax) of the type expressions used in the judgements for the type system. Then I give an overview of the judgements used in the baseline type system. After that, I give some details about wellformed types, type equality and subtyping.

store types	Ψ	::=	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$
state types	σ	::=	$\forall\Delta.\forall\phi.\, R$
type variable contexts	Δ	::=	$\emptyset \mid \Delta, \omega \mid \Delta, \alpha$
regfile types	R	::=	$[r_0 : \tau_0, \ldots, r_n : \tau_n, \texttt{csp} : S_1, \texttt{dsp} : S_2]$
stack types	S	::=	$[\,] \mid \omega \mid \tau :: S$
types	τ	::=	$\alpha \mid \sigma \mid \exists\phi.\tau \mid \texttt{junk} \mid \texttt{int}(e) \mid \texttt{fix} \mid$
			$\tau\ \texttt{xarray}(e) \mid \tau\ \texttt{yarray}(e)$
index expressions	e	::=	$n \mid c \mid e_1 + e_2 \mid -e$
index propositions	P	::=	$e_1 \leq e_2 \mid \neg P \mid P_1 \wedge P_2$
index contexts	ϕ	::=	$\{\} \mid \{b_1, \ldots, b_n \mid P\}$
index variable binding	b	::=	$n : \texttt{int}$
type variables	α		
stack type variables	ω		
index variables	n, k		
constants	c		

Figure 3.1: Type syntax for Featherweight DSP.

3.1.1 Type syntax

Figure 3.1 contains the grammar of type syntax for Featherweight DSP. The basic idea behind the type system is that for a given instruction sequence I the type system will describe the set of valid *machine configurations* or *states* in which it is safe to execute I. Thus, we need to be able to specify the type of a machine configuration. The type of a machine configuration is the type of the store, Ψ, and the type of the register file, R, including the contents of the two stacks. A store type is a finite mapping from named locations to types. And the type of a register file is a finite mapping from each register name, r_i, to the type of the value contained in r_i, plus the mapping for the two special register names `csp` and `dsp` to stack types.

Like in DTAL, we use a restricted form of dependent types to describe the types of integers and the type of arrays. That is, we do not just say that a register contains an integer, we say that a register contains an integer with a specific value. For example, an integer with the value four has type $\texttt{int}(4)$, or more generally, an integer with a value described by the expression e has type $\texttt{int}(e)$. Likewise, an array in X memory with e elements of type τ has type $\tau\ \texttt{xarray}(e)$ (and similarly for Y memory). The expressions e used to specify the length of an array or the value of an integer are called *index expressions*. To keep the type system decidable we restrict index expressions to *Presburger arithmetic* [43] (see Section 3.1.5).

A type variable context, Δ, is a finite set of bound type variables and stack variables.

An index context, ϕ, is a set of bound index variables and a predicate restricting the domain of these (and possibly other) variables. I go into more

detail about index context in Section 3.1.3.

The type τ of a word-sized value is either: a type variable α; a code pointer σ; an existential type over index variables $\exists\phi.\tau$; a fixed-point number `fix`; an integer $\texttt{int}(e)$ with the value e; a pointer τ $\texttt{xarray}(e)$ to an array in X memory; a pointer τ $\texttt{yarray}(e)$ to an array in Y memory; or the non-describing type `junk`.

An index expression e is either: an index variable n; an integer constant c; an addition $e_1 + e_2$; or a negation $-e$. An index proposition is either: an inequality $e_1 \leq e_2$; a logical negation $\neg P$; or a conjunction $P_1 \wedge P_2$. I have kept the formal syntax for index expressions and index propositions minimal. The logical connectives have been chosed arbitrarily, because the choice of which connectives we chose is not important as long as they are functional complete. When we have any two connectives all the other connective can be derived, instead of writing $P_1 \Rightarrow P_2$ we can just write $\neg(P_1 \wedge \neg P_2)$, for instance. Likewise for integer relations, instead of the equality $e_1 = e_2$ we can just write two inequalities $e_1 \leq e_2 \wedge e_2 \leq e_1$. In the rest of the dissertation we shall use such derived forms without further ado.

A state type, σ, describe a code pointer that points to an instruction sequence that requires that the machine configuration can be described by σ when control is transferred to the instruction sequence. A machine configuration is just described by a regfile type, a type variable context, and an index context; for the baseline type system the type of the store will not change once we have established its initial type. Thus, there is no need to pass a store type around. I go into more detail about this in Sections 3.1.7 and 3.2.

A stack type S is either: the empty stack $[\,]$; a stack $\tau :: S$ were the top element has type τ and the rest of the stack has type S; or a stack type variable ω. To describe the two stacks in the abstract machine for Featherweight DSP we could have chosen to have two specialised stack types: one where all the elements are state types for the call-stack, and one where all the elements are integers for the do-stack, but the current formulation of stack types results in more uniform type rules.

Type variables are drawn from a countable infinite set. Likewise are stack type variables, index variables, and label names.

Compared to the type system for DTAL and other type systems, this type syntax does not include several interesting kinds of types: There are no general product types (such as tuples or records), no sum types (such as SML's `datatypes` or DTAL's `choose` types), existential quantification is only allowed over index variables, and universal quantification is only allowed in connection with state types. The reason for these draconian restrictions are that we are not looking for a general purpose type system (for now), we are just looking for a type system that can be used to give type annotations for handwritten DSP assembler code. Thus, I have tried to keep the types to a bare minimum, even if this means loss of generality. Of these restrictions, the lack of product types is the most severe and is actually needed for real-life code, but the baseline type system can be view as a step-stone on the way for the extended type system I

present in Section 3.6. Section 3.6 introduces a novel kind of type construct to remedy the omission of product types.

3.1.2 Overview of Judgements

Figure 3.2 contains an overview of the judgements used in the baseline type system. Some of the judgement consists of a family of judgements one for each syntactic category (e.g., wellformedness judgements), Figure 3.2 only include one judgement from such a family.

The left-hand side of a $\models$ or $\vdash$ judgement is called the *typing context* (not to be confused with either a type variable context or an index context).

The main judgements are those for instructions and instruction sequences. For instructions, the judgement

$$\Delta; \phi_1; \Psi; R_1 \vdash \mathit{ins} \Rightarrow \phi'; R'$$

says that if we execute the instruction *ins* in a machine configuration described by Ψ and R_1, with all type variables bound by Δ and all index variables bound by ϕ_1, then we end up in a machine configuration described by Ψ and R_2, with all type variables bound by Δ and all index variables bound by ϕ_2. That is, we can view $\Delta; \phi_1; \Psi; R_1$ as the *precondition* for *ins* and $\phi_2; R_2$ as a partial *postcondition* (partial because Ψ and Δ are implicitly preserved).

Instruction sequences always end with a control transfer instruction. Hence, in a certain sense an instruction sequence never ends in a machine configuration, the instruction sequence just transfer responsibility to another location (the `halt` instruction can be view as a control transfer to a location that accepts all machine configurations). Thus, for instruction sequences, the judgement:

$$\Delta; \phi; \Psi; R \vdash I$$

just states that if we execute the instruction sequence I in a machine configuration described by Ψ and R, then I will not do anything unsafe. In particular, when I ends by transferring control, then the machine configuration will satisfy the requirements demanded by the location to which control is passed.

3.1.3 Index Contexts

The index context, ϕ, in Figure 3.1 plays an important role in the type system presented in this chapter. Hence, I give a definition of the *domain* of an index context and a definition of the *combination* of two index contexts.

Definition 3.1 (Index context domains)
The domain of an index context ϕ, $\mathbf{dom}(\phi)$, is the set of index variables bound by ϕ:

$$\begin{array}{lcl} \mathbf{dom}(\{\}) & \equiv & \emptyset \\ \mathbf{dom}(\{n_1 : \mathtt{int}, \ldots, n_m : \mathtt{int} \mid P\}) & \equiv & \{n_1, \ldots, n_m\} \end{array}$$

□

Judgement	Description	Defined in:
$\phi \models P$	The index proposition P is satisfied under ϕ.	Section 3.1.5.
$\phi \vdash_{\text{wf}} e$	The index expression e is well-formed under ϕ.	Figure 3.3.
$\phi \vdash \theta : \phi'$	θ is a substitution for ϕ' under ϕ.	Figure 3.4.
$\Delta; \phi \vdash \Theta : \Delta'$	Θ is a substitution for Δ' under Δ and ϕ.	Figure 3.4.
$\Delta; \phi \models \tau_1 \equiv \tau_2$	The types τ_1 and τ_2 are equivalent under $\Delta; \phi$.	Figure 3.5.
$\Delta; \phi \models \tau_1 <: \tau_2$	The type τ_1 is a subtype of the type τ_2. That is, τ_1 can be coerced to τ_2.	Figure 3.6.
$\Delta; \phi \models R_1 <: R_2$	R_1 can be coerced to R_2.	Figure 3.6.
$\phi; \Psi; R \vdash v : \tau$	The value v has type τ.	Figure 3.7.
$\phi; \Psi; R \vdash \mathit{aexp} : \tau$	The arithmetic expression *aexp* has type τ.	Figure 3.7.
$\Delta; \phi; \Psi; R \vdash \mathit{sins} \Rightarrow R'$	The small instruction *sins* returns the regfile type R' which contains at most one binding.	Figure 3.8.
$\Delta; \phi; \Psi; R \vdash \mathit{ins} \Rightarrow \phi'; R'$	The instruction *ins* transforms the regfile type R to R' and the index context ϕ to ϕ'.	Figure 3.9.
$\Delta; \phi; \Psi; R \vdash B \Rightarrow \phi'; R'$	The do-body B transforms the regfile type R to R' and the index context ϕ to ϕ'.	Figure 3.11.
$\Delta; \phi; \Psi; R \vdash I$	The instruction sequence I is well-typed.	Figure 3.11.
$\vdash (\ell_i, \mathit{prog})$	The program *prog* is well-typed and it is safe the start the execution at the instruction sequence at the label ℓ_i.	Figure 3.12.

Figure 3.2: Overview of judgements for the baseline type system.

$$\phi \vdash_{\text{wf}} c \qquad \frac{n \in \mathbf{dom}(\phi)}{\phi \vdash_{\text{wf}} n} \qquad \frac{\phi \vdash_{\text{wf}} e_1 \qquad \phi \vdash_{\text{wf}} e_2}{\phi \vdash_{\text{wf}} e_1 + e_2} \qquad \frac{\phi \vdash_{\text{wf}} e}{\phi \vdash_{\text{wf}} -e}$$

$$\frac{\phi \vdash_{\text{wf}} e_1 \qquad \phi \vdash_{\text{wf}} e_2}{\phi \vdash_{\text{wf}} e_1 \leq e_2} \qquad \frac{\phi \vdash_{\text{wf}} P}{\phi \vdash_{\text{wf}} \neg P} \qquad \frac{\phi \vdash_{\text{wf}} P_1 \qquad \phi \vdash_{\text{wf}} P_2}{\phi \vdash_{\text{wf}} P_1 \wedge P_2}$$

$$\frac{\alpha \in \Delta}{\Delta; \phi \vdash_{\text{wf}} \alpha} \qquad \Delta; \phi \vdash_{\text{wf}} \texttt{junk} \qquad \frac{\phi \vdash_{\text{wf}} e}{\Delta; \phi \vdash_{\text{wf}} \texttt{int}(e)} \qquad \frac{\Delta; \phi \vdash_{\text{wf}} \tau \qquad \phi \vdash_{\text{wf}} e}{\Delta; \phi \vdash_{\text{wf}} \tau\ \texttt{xarray}(e)}$$

$$\frac{\phi_1 \vdash_{\text{wf}} \phi_2 \quad \Delta_1 \cap \Delta_2 = \emptyset \quad \mathbf{dom}(\phi_1) \cap \mathbf{dom}(\phi_2) = \emptyset \quad \Delta_1 \cup \Delta_2; \phi_1 \wedge \phi_2 \vdash_{\text{wf}} R}{\Delta_1; \phi_1 \vdash_{\text{wf}} \forall \Delta_2. \forall \phi_2.\, R}$$

$$\Delta; \phi \vdash_{\text{wf}} [] \qquad \frac{\omega \in \Delta}{\Delta; \phi \vdash_{\text{wf}} \omega} \qquad \frac{\Delta; \phi \vdash_{\text{wf}} \tau \quad \Delta; \phi \vdash_{\text{wf}} S}{\Delta; \phi \vdash_{\text{wf}} \tau :: S}$$

$$\phi \vdash_{\text{wf}} \{\} \qquad \frac{\phi \wedge \{b_1, \ldots, b_n\} \vdash_{\text{wf}} P}{\phi \vdash_{\text{wf}} \{b_1, \ldots, b_n \mid P\}}$$

$$\frac{\Delta; \phi \vdash_{\text{wf}} \tau_0 \quad \cdots \quad \Delta; \phi \vdash_{\text{wf}} \tau_n \qquad \Delta; \phi \vdash_{\text{wf}} S_1 \quad \Delta; \phi \vdash_{\text{wf}} S_2}{\Delta; \phi \vdash_{\text{wf}} [r_0 : \tau_0, \ldots, r_n : \tau_n, \texttt{csp} : S_1, \texttt{dsp} : S_2]}$$

Figure 3.3: Well-formed index expressions, propositions, types, index contexts, and register files.

Definition 3.2 (Combining index contexts)
Given two index contexts ϕ_1 and ϕ_2 where $\mathbf{dom}(\phi_1)$ and $\mathbf{dom}(\phi_2)$ are disjoint, define the combination, $\phi_1 \wedge \phi_2$, of ϕ_1 and ϕ_2 as:

$$\begin{array}{lcl} \{\} \wedge \phi & \equiv & \phi \\ \phi \wedge \{\} & \equiv & \phi \\ \{b_{11}, \ldots, b_{1n} \mid P_1\} \wedge \{b_{21}, \ldots, b_{2m} \mid P_2\} & \equiv & \{b_{11}, \ldots, b_{1n}, b_{21}, \ldots, b_{2m} \mid P_1 \wedge P_2\} \end{array}$$ □

We use $\phi \wedge P$ as a shorthand for $\phi \wedge \{ \mid P\}$, and $\phi \wedge \{b_1, \ldots, b_n\}$ as a shorthand for $\phi \wedge \{b_1, \ldots, b_n \mid \mathit{true}\}$.

3.1.4 Well-formed Index Contexts, Types, Expressions and Propositions

Figure 3.3 presents the judgements for forming well-formed index expressions, propositions, types, index contexts, and register files.

The well-formed relations are defined with respect to an index context, ϕ, or a type variable context and an index context, $\Delta; \phi$. The relations basically state that all the type variables and index variables in a term (index expression, proposition, etc.) are bound in the contexts.

In the type rule for well-formed state types in Figure 3.3, we combine two index contexts, $\phi_1 \wedge \phi_2$ (see Definition 3.2 in the previous section). This rule requires that the domains of the two index context are disjunct and that there is no overlap of bound type variables. If this requirement is not satisfied the bound index variables, type variables, and stack variables must be suitably renamed (α-converted).

3.1.5 Solving Constraints

The satisfiability relation $\phi \models P$ means that the formula $(\phi)P$ is satisfiable in the domain of integers. The formula $(\phi)P$ is defined as follows:

$$\begin{array}{lcl} (\{\})P & \equiv & P \\ (\{n_1 : \texttt{int}, \ldots, n_m : \texttt{int} \mid P_1\})P_2 & \equiv & \forall n_1, \ldots, n_m.(P_1 \Rightarrow P_2) \end{array}$$

Given the satisfiability relation we can now define what a *consistent* index context is.

Definition 3.3 (Consistent index contexts)
An index context ϕ is *consistent* if and only if it is not possible to derive $\phi \models \mathit{false}$; otherwise it is *inconsistent*. We use the notation $\vdash_c \phi$ to say that ϕ is consistent.□

It is worth noting that the constraints are defined in *Presburger arithmetic* [43], also called theory of *intergers with addition and order* [25, page 354]. Presburger arithmetic is decidable a deciable theory, although the decision procedure has a super-exponential complexity, $O(2^{2^{2^{pn}}})$, in the size of the formula that is checked. If we allow multiplication as well as addition then we have *number theory*, which is undecidable—Gödel's famous Incompleteness Theorem [18].

3.1.6 Substitutions

Substitutions are defined in the standard manner, that is, they are capture-avoiding and we silently allow renaming of bound variables (α-conversion) in types. Given a type term t, for example a plain type or a regfile type, we use the notation $t[\theta]$ for the result of applying θ to t. Substitutions are finite mappings:

$$\begin{array}{lcll} \text{index variable substitutions} & \theta & ::= & [] \mid \theta[n \mapsto e] \\ \text{type and stack variable substitutions} & \Theta & ::= & [] \mid \Theta[\alpha \mapsto \tau] \mid \Theta[\omega \mapsto S] \end{array}$$

Figure 3.4 introduces two judgements $\phi \vdash \theta : \phi'$ and $\Delta; \phi \vdash \Theta : \Delta'$ for substitutions and presents the rules for deriving these judgements.

The judgement $\phi \vdash \theta : \phi'$, where ϕ' is $\{b_1, \ldots, b_n \mid P'\}$, means that θ replaces all index variables in $\mathbf{dom}(\phi')$ by index expressions involving only variables in $\mathbf{dom}(\phi)$, and $P'[\theta]$ holds under ϕ. Similarly, the judgement $\Delta; \phi \vdash \Theta : \Delta'$ means that Θ replaces all type and stack variables in Δ' with types and stack types wellformed in $\Delta; \phi$. Thus, both kinds of substitutions preserve well-formedness.

$$\phi \vdash [] : \{\} \qquad \frac{\phi \vdash_{\text{wf}} e_1 \quad \cdots \quad \phi \vdash_{\text{wf}} e_m \qquad \theta = [n_1 \mapsto e_1, \ldots, n_m \mapsto e_m] \qquad \phi \models P[\theta]}{\phi \vdash \theta : \{n_1 : \texttt{int}, \ldots, n_m : \texttt{int} \mid P\}}$$

$$\Delta; \phi \vdash [] : \emptyset \qquad \frac{\Delta; \phi \vdash_{\text{wf}} \tau \qquad \Delta; \phi \vdash \Theta : \Delta'}{\Delta; \phi \vdash \Theta[\alpha \mapsto \tau] : \Delta', \alpha} \qquad \frac{\Delta; \phi \vdash_{\text{wf}} S \qquad \Delta; \phi \vdash \Theta : \Delta'}{\Delta; \phi \vdash \Theta[\omega \mapsto S] : \Delta', \omega}$$

Figure 3.4: Substitutions

That is, given consistent contexts ϕ and ϕ', if $\Delta; \phi \vdash_{\text{wf}} R$ and $\phi' \vdash \theta : \phi$ then $\Delta; \phi \vdash_{\text{wf}} R[\theta]$, and similarly for type variable substitutions.

For both kinds of substitutions, all parts of a substitution are performed in parallel. That is, if we have the type variable substitution

$$\Theta = [a \mapsto b, b \mapsto c]$$

and the regfile type

$$R = [r_0 : a, r_1 : b]$$

and we perform the substitution $R[\Theta]$ then we get the regfile type $[r_0 : b, r_1 : c]$ and not $[r_0 : c, r_1 : c]$.

3.1.7 Type Equality

Figure 3.5 presents the rules of the equality relation for types and for stack types. Equality of two types τ_1 and τ_2 is only defined with respect to a type variable context Δ and an index variable context ϕ.

Equality for state types (**state-eq**) and existential types (**exist-eq**) is defined in terms of the subtyping relation which is presented in the following section. In addition to these two rules, the only interesting thing to notice about the rules in Figure 3.5 is how the typing context $\Delta; \phi$ is passed through the rules and used to determine whether two index expressions are equal.

Lemma 3.1 (Equivalence relation)
The relation $\equiv$ denotes a family of equivalence relations indexed by a type variable context Δ and an index variable context ϕ. That is, the following three properties holds:

Reflexive: *If $\Delta; \phi \vdash_{\text{wf}} \tau$ then $\Delta; \phi \models \tau \equiv \tau$.*

Transitive: *If both $\Delta; \phi \models \tau_1 \equiv \tau_2$ and $\Delta; \phi \models \tau_2 \equiv \tau_3$ then $\Delta; \phi \models \tau_1 \equiv \tau_3$*

Symmetric: *If $\Delta; \phi \models \tau_1 \equiv \tau_2$ then $\Delta; \phi \models \tau_2 \equiv \tau_1$.* □

(**junk-eq**) $$\Delta; \phi \models \texttt{junk} \equiv \texttt{junk}$$

(**fix-eq**) $$\Delta; \phi \models \texttt{fix} \equiv \texttt{fix}$$

(**tvar-eq**) $$\frac{\alpha_1 \in \Delta \qquad \alpha_2 \in \Delta \qquad \alpha_1 = \alpha_2}{\Delta; \phi \models \alpha_1 \equiv \alpha_2}$$

(**exist-eq**) $$\frac{\Delta; \phi \models \exists \phi_1.\tau_1 <: \exists \phi_2.\tau_2 \qquad \Delta; \phi \models \exists \phi_2.\tau_2 <: \exists \phi_1.\tau_1}{\Delta; \phi \models \exists \phi_1.\tau_1 \equiv \exists \phi_2.\tau_2}$$

(**int-eq**) $$\frac{\phi \models e_1 = e_2}{\Delta; \phi \models \texttt{int}(e_1) \equiv \texttt{int}(e_2)}$$

(**array-eq**) $$\frac{\Delta; \phi \models \tau_1 \equiv \tau_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \tau_1\ \texttt{xarray}(e_1) \equiv \tau_2\ \texttt{xarray}(e_2)}$$

(**state-eq**) $$\frac{\Delta; \phi \models \forall \Delta_1.\forall \phi_1.\, R_1 <: \forall \Delta_2.\forall \phi_2.\, R_2 \qquad \Delta; \phi \models \forall \Delta_2.\forall \phi_2.\, R_2 <: \forall \Delta_1.\forall \phi_1.\, R_1}{\Delta; \phi \models \forall \Delta_1.\forall \phi_1.\, R_1 \equiv \forall \Delta_2.\forall \phi_2.\, R_2}$$

(**stvar-eq**) $$\frac{\omega_1 \in \Delta \qquad \omega_2 \in \Delta \qquad \omega_1 = \omega_2}{\Delta; \phi \models \omega_1 \equiv \omega_2}$$

(**empty-eq**) $$\Delta; \phi \models [] \equiv []$$

(**stack-eq**) $$\frac{\Delta; \phi \models \tau_1 \equiv \tau_2 \qquad \Delta; \phi \models S_1 \equiv S_2}{\Delta; \phi \models \tau_1 :: S_1 \equiv \tau_2 :: S_2}$$

Figure 3.5: Type equality $\Delta; \phi \models \tau_1 \equiv \tau_2$.

Proof (sketch) Standard proof by induction over the depth of derivation trees. The proof assumes that equality for index expressions $\phi \models e_1 = e_2$ is an equivalence relation (which it is). ∎

3.1.8 Subtyping

Figure 3.6 contains the rules for the subtyping relation for types, for stack types, and for regfile types. Like in the previous section, whether one type is a subtype of another type is only defined with respect to a type variable context Δ and an index variable context ϕ.

The interesting rules in Figure 3.6 are: the rule for array types (**array-sub**), the rules for existential types (**existl-sub**) and (**existr-sub**), the rule for state types (**state-sub**), and the rule for regfile types (**regs-sub**).

(**junk-sub**) $$\Delta; \phi \models \tau <: \texttt{junk}$$

(**fix-sub**) $$\Delta; \phi \models \texttt{fix} <: \texttt{fix}$$

(**tvar-sub**) $$\frac{\alpha_1 \in \Delta \qquad \alpha_2 \in \Delta \qquad \alpha_1 = \alpha_2}{\Delta; \phi \models \alpha_1 <: \alpha_2}$$

(**existl-sub**) $$\frac{\mathbf{dom}(\phi) \cap \mathbf{dom}(\phi') = \emptyset \qquad \Delta; \phi \wedge \phi' \models \tau_1 <: \tau_2}{\Delta; \phi \models \exists \phi'.\tau_1 <: \tau_2}$$

(**existr-sub**) $$\frac{\phi \vdash \theta : \phi' \qquad \Delta; \phi \models \tau_1 <: \tau_2[\theta]}{\Delta; \phi \models \tau_1 <: \exists \phi'.\tau_2}$$

(**int-sub**) $$\frac{\phi \models e_1 = e_2}{\Delta; \phi \models \texttt{int}(e_1) <: \texttt{int}(e_2)}$$

(**array-sub**) $$\frac{\Delta; \phi \models \tau_1 \equiv \tau_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \tau_1\ \texttt{xarray}(e_1) <: \tau_2\ \texttt{xarray}(e_2)}$$

(**state-sub**) $$\frac{\begin{array}{c}\Delta \cap \Delta_2 = \emptyset \qquad \mathbf{dom}(\phi) \cap \mathbf{dom}(\phi_2) = \emptyset \\ \phi_2 \vdash \theta : \phi_1 \qquad \Delta_2; \phi_2 \vdash \Theta : \Delta_1 \\ \Delta, \Delta_2; \phi \wedge \phi_2 \models R_2 <: R_1[\Theta][\theta]\end{array}}{\Delta; \phi \models \forall \Delta_1.\forall \phi_1.\, R_1 <: \forall \Delta_2.\forall \phi_2.\, R_2}$$

(**stvar-sub**) $$\frac{\omega_1 \in \Delta \qquad \omega_2 \in \Delta \qquad \omega_1 = \omega_2}{\Delta; \phi \models \omega_1 <: \omega_2}$$

(**empty-sub**) $$\Delta; \phi \models [] <: []$$

(**stack-sub**) $$\frac{\Delta; \phi \models \tau_1 <: \tau_2 \qquad \Delta; \phi \models S_1 <: S_2}{\Delta; \phi \models \tau_1 :: S_1 <: \tau_2 :: S_2}$$

(**regs-sub**) $$\frac{\begin{array}{c}\Delta; \phi \models R_1(\texttt{csp}) <: R_2(\texttt{csp}) \qquad \Delta; \phi \models R_1(\texttt{dsp}) <: R_2(\texttt{dsp}) \\ \Delta; \phi \models R_1(r) <: R_2(r) \qquad \text{for all } r \text{ in } R_2\end{array}}{\Delta; \phi \models R_1 <: R_2}$$

Figure 3.6: Subtype relation $\Delta; \phi \models \tau_1 <: \tau_2$.

The rule for array types says that the `xarray` and the `yarray` type constructors are *invariant* in the type of the elements. The reason for this is to ensure that each store location is associated with at most one type. That is, the type system follow the type invariance principle (see Section 1.4.3).

The rule for left elimination of existential types (**existl-sub**) corresponds to the logical implication

$$\forall x.P \Rightarrow \exists x.P.$$

Or, informally, if we can prove that no matter how we instantiate the index variables in ϕ' (moving ϕ' to the left-hand side of $\models$ corresponds to universal quantification) the subtyping between τ_1 and τ_2 holds, then it is safe to eliminate the existential quantifier. The rule (**existr-sub**) for elimination of existential types on the right-hand side of $<:$ uses the substitutions as described in Section 3.1.6 to check if the index variables bound by ϕ' can be instantiated in τ_2 such that the subtyping between τ_1 and τ_2 holds.

The typing rule for regfile types (**regs-sub**) is just the standard point wise subtype rule for extensible record types, see for example [42, Chapter 15]. The rule for state types (**state-sub**) is more interesting. At first, we might think that R_1 and R_2 have been accidentally swapped in the premise for the rule, but that is not the case. Recall that state type are used for code pointers; we can think of a code pointer as a function in continuation passing style that takes a register file as argument and never returns (it just calls the continuation passed as an argument). That is, instead of using the syntax $\forall\Delta.\forall\phi.\, R$ for state types, we could use the conventional notation for function types using an arrow: $\forall\Delta.\forall\phi.\, R \rightarrow \bullet$ (where $\bullet$ means that the function does not return). Now the type rule for state types makes more sense, because the regfile type appears in a *contravariant* position.

For completeness, we note that the subtype relation is a partial order with respect to the equality relation defined in Section 3.1.7.

Lemma 3.2 (Partial ordering)
The relation $<:$ denotes a family of partial orderings on two types indexed by a type variable context Δ and an index variable context ϕ. That is, the following properties hold:

Reflexive: *If* $\Delta;\phi \models \tau_1 \equiv \tau_2$ *then* $\Delta;\phi \models \tau_1 <: \tau_2$.

Transitive: *If both* $\Delta;\phi \models \tau_1 <: \tau_2$ *and* $\Delta;\phi \models \tau_2 <: \tau_3$ *then* $\Delta;\phi \models \tau_1 <: \tau_3$.

Anti-symmetric *If* $\Delta;\phi \models \tau_1 <: \tau_2$ *and* $\Delta;\phi \models \tau_2 <: \tau_1$ *then* $\Delta;\phi \models \tau_1 \equiv \tau_2$. □

Proof (sketch) Again, standard proof by induction of the depth of derivation trees. ∎

$$\textbf{(int)}\quad \phi;\Psi;R \vdash i : \texttt{int}(i) \qquad \textbf{(fix)}\quad \phi;\Psi;R \vdash f : \texttt{fix}$$

$$\textbf{(lab)}\quad \frac{\Psi(\ell) = \tau}{\phi;\Psi;R \vdash \ell : \tau} \qquad \textbf{(reg)}\quad \frac{R(r) = \tau}{\phi;\Psi;R \vdash r : \tau}$$

$$\textbf{(add-fix)}\quad \frac{R(r_1) = \texttt{fix} \qquad R(r_2) = \texttt{fix}}{\phi;\Psi;R \vdash r_1 \texttt{ + } r_2 : \texttt{fix}}$$

$$\textbf{(add-int)}\quad \frac{R(r_1) = \texttt{int}(e_1) \qquad R(r_2) = \texttt{int}(e_2)}{\phi;\Psi;R \vdash r_1 \texttt{ + } r_2 : \texttt{int}(e_1 + e_2)}$$

$$\textbf{(add-xarr1)}\quad \frac{R(r_1) = \texttt{int}(e_1) \qquad \phi \models 0 \leq e_1 \qquad R(r_2) = \tau\ \texttt{xarray}(e_2)}{\phi;\Psi;R \vdash r_1 \texttt{ + } r_2 : \tau\ \texttt{xarray}(e_2 - e_1)}$$

$$\textbf{(add-xarr2)}\quad \frac{R(r_2) = \texttt{int}(e_2) \qquad \phi \models 0 \leq e_2 \qquad R(r_1) = \tau\ \texttt{xarray}(e_1)}{\phi;\Psi;R \vdash r_1 \texttt{ + } r_2 : \tau\ \texttt{xarray}(e_1 - e_2)}$$

$$\textbf{(mult-fix)}\quad \frac{R(r_1) = \texttt{fix} \qquad R(r_2) = \texttt{fix}}{\phi;\Psi;R \vdash r_1 \texttt{ * } r_2 : \texttt{fix}}$$

$$\textbf{(mult-int)}\quad \frac{R(r_1) = \texttt{int}(e_1) \qquad R(r_2) = \texttt{int}(e_2)}{\phi;\Psi;R \vdash r_1 \texttt{ * } r_2 : \texttt{int}(e_1 \cdot e_2)}$$

Figure 3.7: Typing of values and arithmetic expressions.

3.2 Baseline Type System

The previous section described the syntax, the equality relation, and subtyping relation of types. With these basic notions in place we are now ready for the more Featherweight DSP specific parts. This section describes the baseline type system for Featherweight DSP. That is, typing judgements for values, arithmetic expressions, small instructions, instructions, and instruction sequences. The type system presented in this section is largely the type system for DTAL adapted to Featherweight DSP.

3.2.1 Typing of Values and Arithmetic Expressions

Figure 3.7 show the typing rules for values and arithmetic expressions. Strictly speaking, this is two different judgements but we shall just treat them as one and it should be clear from the context which one we use. The typing context for these judgement is just an index context ϕ, a store type Ψ, and a regfile type R.

The typing rules for arithmetic expression show how index expressions are used to track the value of an source language expression. These rules are not syntax directed because, the syntax alone does not determine the type of an expression or value. Thus, we need multiple type rules for the same syntactic class of expression. The rules for `yarray` are not shown as they are similar to the rules for `xarray` (**add-xarr1**) and (**add-xarr2**).

The interesting rules in Figure 3.7 are the rule for integer multiplication (**mult-int**) and the rules for pointer arithmetic (**add-xarr1**) and (**add-xarr2**). In the rule (**mult-int**) for integer multiplication it looks like we are forming an invalid index expression in the conclusion, and indeed we are. We shall allow this because there are a lot of special cases that are convenient to allow: such as if e_1 or e_2 is a constant, or if the expression in the conclusion never ends up in a Presburger proposition that need to be checked for satisfiability. Instead of the type $\texttt{int}(e_1 \cdot e_2)$ we could use an existential type $\exists\{k : \texttt{int}\}.\texttt{int}(k)$. This type convey that there exists an integer k which is the result of multiplying the two integers expressions e_1 and e_2, but we have not other information about k than it exists. The type rules (**add-xarr1**) and (**add-xarr2**) for shows that we only allow restricted pointer arithmetic, only addition with an positive integer is allowed. The intuition behind the rules for pointer arithmetic is that if you have pointer to an array with e_1 elements and you add e_2 to this pointer, then you have a new pointer, this time to an array with $e_1 - e_2$ elements. Here it is worth noting that the array type $\tau\ \texttt{xarray}(-2)$ is a perfectly valid and wellformed type. The interpretation of an array type with negative size, is that you have incremented the pointer past the end of the array, which valid but you are no longer allowed to read from or write to memory using that pointer, as we shall see in the following section.

3.2.2 Typing of Instructions

Figure 3.8 show the rules for small instructions. That is, instructions that can be put in parallel to form a composite instruction. The typing rule for composite instructions is in Figure 3.9 which is described in the following. Again, only the rules for `xarray` is shown.

Similar to the rules for arithmetic expressions, some of the rules in Figure 3.8 (**incr-fix**), (**incr-int**), and (**incr-xarr**) are not syntax directed, and we need multiple rules for the same syntactic class of expression. The interesting rules in Figure 3.8 are the typing rule (**read**) for reading from memory and the rule (**write**) for writing to memory. These are the rules that ensure memory safety, the rule (**read**) only allows reads from memory if we know that we are within the bounds of an array, and similar the rule (**write**) only allows that we store values within the bounds of an array. These rules also show why only increments are allowed to pointers, because the rules (**read**) and (**write**) store only checks that we have not moved the pointer past the end of the array. If we allowed a pointer to an array be decremented, then the pointer could be moved before the

$$\textbf{(read)}\quad \frac{\Psi; R \vdash r_2 : \tau\ \texttt{xarray}(e) \qquad \phi \models e > 0}{\Delta; \phi; \Psi; R \vdash r_1\ \texttt{=}\ \texttt{xmem[}r_2\texttt{]} \Rightarrow [r_1 : \tau]}$$

$$\textbf{(write)}\quad \frac{\begin{array}{c}\Psi; R \vdash r_1 : \tau_1\ \texttt{xarray}(e) \\ \phi \models e > 0 \qquad \Psi; R \vdash r_2 : \tau_2 \qquad \Delta; \phi \models \tau_2 <: \tau_1\end{array}}{\Delta; \phi; \Psi; R \vdash \texttt{xmem[}r_1\texttt{]}\ \texttt{=}\ r_2 \Rightarrow []}$$

$$\textbf{(incr-fix)}\quad \frac{R(r) = \texttt{fix} \qquad \phi; \Psi; R \vdash \mathit{aexp} : \texttt{fix}}{\Delta; \phi; \Psi; R \vdash r\ \texttt{+=}\ \mathit{aexp} \Rightarrow [r : \texttt{fix}]}$$

$$\textbf{(incr-int)}\quad \frac{R(r) = \texttt{int}(e_1) \qquad \phi; \Psi; R \vdash \mathit{aexp} : \texttt{int}(e_2)}{\Delta; \phi; \Psi; R \vdash r\ \texttt{+=}\ \mathit{aexp} \Rightarrow [r : \texttt{int}(e_1 + e_2)]}$$

$$\textbf{(incr-xarr)}\quad \frac{R(r) = \tau\ \texttt{xarray}(e_1) \qquad \phi; \Psi; R \vdash \mathit{aexp} : \texttt{int}(e_2) \qquad \phi \models 0 \leq e_2}{\Delta; \phi; \Psi; R \vdash r\ \texttt{+=}\ \mathit{aexp} \Rightarrow [r : \tau\ \texttt{xarray}(e_1 - e_2)]}$$

$$\textbf{(assign)}\quad \frac{\phi; \Psi; R \vdash \mathit{aexp} : \tau}{\Delta; \phi; \Psi; R \vdash r\ \texttt{=}\ \mathit{aexp} \Rightarrow [r : \tau]}$$

Figure 3.8: Type rules for small instructions.

beginning of the array. The rules could be adapted to allow this, we would just have to instrument the array types with and extra index expression. Also, it is important to note that the rules in Figure 3.8 only return a regfile with at most one binding, namely for the register that has been modified.

Figure 3.9 shows the typing rules for instructions. Here the rules are more interesting. The first rule (**eelim**) for unpacking existential types is really just a coercion rule combined with the subtype rule (**existl-sub**) from Figure 3.6.

The rule (**comp**) is for typing composite instructions, and uses the function UniqDef from Section 2.3 to ensure that there are no race conditions. This check is not strictly needed in the rule, because we only work with wellformed programs, that is programs that satisfy UniqDef, but the check is here for clarity. Also the rule crucially relies on the property that no race conditions can occur, which is enforced by UniqDef. The resulting regfile R' is a composition of the original regfile R and all the simple regfile types yielded by the small instructions.

The rule for conditional jumps (**beq**) is the first time we see a rule for a control transfer instruction (there are, of cause, similar rules for the conditional jump instructions, but these rules are left out from this presentation). In the rule we first check that r contains an integer value and that v is a code address (either directly with a label or through a register), then we check that it is safe to jump to the code pointed to by v if r contains the integer value zero. That is, we see if we can find two substitutions, one for type variables and one for the index

(**eelim**)
$$\frac{\mathbf{dom}(\phi_1) \cap \mathbf{dom}(\phi_2) = \emptyset \qquad \Delta; \phi_1 \wedge \phi_2; \Psi; R\{r : \tau\} \vdash ins \Rightarrow \phi'; R'}{\Delta; \phi_1; \Psi; R\{r : \exists \phi_2.\tau\} \vdash ins \Rightarrow \phi'; R'}$$

(**comp**)
$$\frac{\begin{array}{c}\textsc{UniqDef}(sins_1, \ldots, sins_n) \\ \Delta; \phi; \Psi; R \vdash sins_1 \Rightarrow R_1 \quad \cdots \quad \Delta; \phi; \Psi; R \vdash sins_n \Rightarrow R_n \\ R' = R + R_1 + \ldots + R_n\end{array}}{\Delta; \phi; \Psi; R \vdash sins_1 \,; \ldots ; sins_n \Rightarrow \phi; R'}$$

(**beq**)
$$\frac{\begin{array}{c}\Psi; R \vdash r : \mathtt{int}(e) \\ \Psi; R \vdash v : \ \forall \Delta'.\forall \phi'.\, R' \\ \phi \wedge e = 0 \vdash \theta : \phi' \qquad \Delta; \phi \wedge e = 0 \vdash \Theta : \Delta' \\ \Delta; \phi \wedge e = 0 \models R <: R'[\Theta][\theta]\end{array}}{\Delta; \phi; \Psi; R \vdash \mathtt{beq}\ r,\ v \Rightarrow \phi \wedge e \neq 0; R}$$

(**call**)
$$\frac{\begin{array}{c}R_1(\mathtt{csp}) = S \\ \Psi; R_1 \vdash v : \forall \Delta_2.\forall \phi_2.\, R_2 \\ \phi_1 \vdash \theta : \phi_2 \qquad \Delta_1; \phi_1 \vdash \Theta : \Delta_2 \\ \Delta_1; \phi_1 \models R_1\{\mathtt{csp} : \forall \emptyset.\forall \phi_3.\, R_3 :: S\} <: R_2[\Theta][\theta]\end{array}}{\Delta_1; \phi_1; \Psi; R_1 \vdash \mathtt{call}(v) \Rightarrow \phi_3; R_3}$$

(**do**)
$$\frac{\begin{array}{c}\Psi; R_1 \vdash v : \mathtt{int}(e) \qquad \phi_1 \models e > 0 \\ R_1(\mathtt{dsp}) = S \\ k_1 \in \mathbf{dom}(\phi_2) \qquad \phi_2 \models 0 \leq k_1 < e \\ R_2(\mathtt{dsp}) = \mathtt{int}(k_1) :: S \\ \phi_1 \vdash \theta_1 : \phi_2 \\ \Delta; \phi_1 \models R_1\{\mathtt{dsp} : \mathtt{int}(0) :: S\} <: R_2[\theta_1] \\ \Delta; \phi_2; \Psi; R_2 \vdash B \Rightarrow \phi_3; R_3\{\mathtt{dsp} : \mathtt{int}(k_1) :: S\} \\ \phi_3' = \phi_3 \wedge \{k_2 : \mathtt{int} \mid k_1 < e - 1 \wedge k_2 = k_1 + 1\} \qquad k_2 \notin \mathbf{dom}(\phi_3) \\ \phi_3' \vdash \theta_2 : \phi_2 \\ \Delta; \phi_3' \models R_3\{\mathtt{dsp} : \mathtt{int}(k_2) :: S\} <: R_2[\theta_2]\end{array}}{\Delta; \phi_1; \Psi; R_1 \vdash \mathtt{do}(v)\ \{B\} \Rightarrow \phi_3 \wedge k_1 = e - 1; R_3\{\mathtt{dsp} : S\}}$$

(**enddo**)
$$\frac{R(\mathtt{dsp}) = \mathtt{int}(e) :: S}{\Delta; \phi; \Psi; R \vdash \mathtt{enddo} \Rightarrow \phi; R\{\mathtt{dsp} : S\}}$$

Figure 3.9: Type rules for instructions.

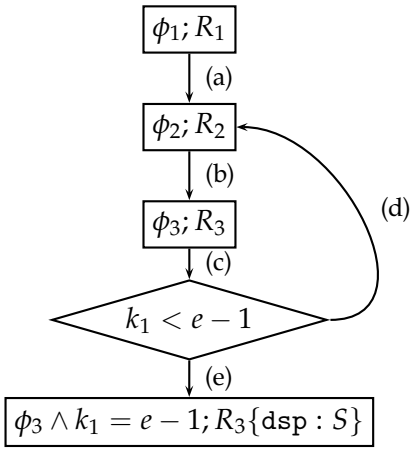


Figure 3.10: Diagram for explaining the (**do**) rule.

variables, so that R is a subtype of R'; otherwise we know the integer value e in r is different from zero and we update the index context to record this.

The rule for procedure calls (**call**) is also a rule for a control transfer instruction, and the rule follows the same pattern as the (**beq**) rule. The only change is that we have to push the return address, which is a code pointer, to the call-stack and this is reflected in the type rule. One subtlety in this rule is that the state type pushed on the stack must not bind any type variables, because I only want to introduce new type variables at named code locations.

The rule for `do`-loops is more involved than the other rules, and is described separately in the following. The rule for the `enddo` instruction (**enddo**) on the other hand is rather simple. Remember that the `enddo` is used when we have branched out of a `do`-loop and need to clean up the do-stack, thus we simply pop the top of the do-stack.

The type rule for `do`-loops

The rule for `do`-loops in Figure 3.9 is complicated. This section explains the rule by dissecting the premise, piece by piece. Figure 3.10 gives diagrammatic aid for the explanation.

To check a `do` instruction:

`do(`v`) {`B`}`

in the typing context $\Delta; \phi_1; \Psi; R_1$ we proceed as follows:

- First, we check that the loop count v is an integer value and that it is strictly greater than zero (the latter requirement is inherited from the real custom DSP):

$$\Psi; R_1 \vdash v : \texttt{int}(e) \qquad \text{and} \qquad \phi_1 \models e > 0.$$

- Then we must guess a typing context for type checking the loop body B. That is, we must find an index context ϕ_2 and a regfile type R_2 (the type variable context Δ and the store type Ψ remains fixed). The typing context must record that the loop counter is on top of the do-stack, that this counter has a value between zero and the loop count e, and except for the top of the do-stack the do-stack must have the same contents as when we enter the loop:

$$k_1 \in \mathbf{dom}(\phi_2) \qquad \text{and} \qquad \phi_2 \models 0 \leq k_1 < e$$
$$\text{and } R_1(\texttt{dsp}) = S \qquad \text{and} \qquad R_2(\texttt{dsp}) = \texttt{int}(k_1) :: S$$

  As a side note, you might find it helpful to think of the typing context ϕ_2 and R_2 as a *loop invariant*.

- Next, we check that when we enter the do instruction the starting context ϕ_1 and R_1 is compatible with the loop body context ϕ_2 and R_2 if we push a zero on the do-stack, that is (a) in Figure 3.10:

$$\phi_1 \vdash \theta_1 : \phi_2 \qquad \text{and} \qquad \Delta; \phi_1 \models R_1\{\texttt{dsp} : \texttt{int}(0) :: S\} <: R_2[\theta_1]$$

  That is, the loop counter starts at zero and counts up to the loop count.

- Then we check the body of the loop to get the context ϕ_3 and R_3 at the end of the loop body, that is (b) in Figure 3.10:

$$\Delta; \phi_2; \Psi; R_2 \vdash B \Rightarrow \phi_3; R_3\{\texttt{dsp} : \texttt{int}(k_1) :: S\}$$

  The typing rule for the body B is described in Section 3.2.3.

- After that, we check that it is safe to jump back to the top of the loop body in all iterations but the last one(i.e., that the context ϕ_3' and R_3, with an updated do-stack, is compatible with the loop body context ϕ_2 and R_2. Also, we increment the loop counter on the top of the do-stack, that is (c) and (d) in Figure 3.10:

$$\phi_3' = \phi_3 \wedge \{k_2 : \texttt{int} \mid k_1 < e - 1 \wedge k_2 = k_1 + 1\} \qquad k_2 \notin \mathbf{dom}(\phi_3)$$
$$\phi_3' \vdash \theta_2 : \phi_2 \qquad \text{and} \qquad \Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: S\} <: R_2[\theta_2]$$

  Incrementing the loop counter is done by introducing a fresh index variable k_2 and then substituting all occurrences of k_1 by k_2 in R_2.

- Finally, we end up with the resulting typing context where we have popped the loop counter off the do-stack and we know that if we reach this point in the program then k_1 must have the value $e - 1$, that is (e) in Figure 3.10:

$$\phi_3 \wedge k_1 = e - 1; R_3\{\texttt{dsp} : S\}$$

  Remember that k_1 is the value of the loop counter at the *entry* of the loop which is why it is $e - 1$ and not e.

$$\textbf{(jmp)}\quad \frac{\Psi; R \vdash v : \ \forall\Delta'.\forall\phi'.\, R' \qquad \phi \vdash \theta : \phi' \qquad \Delta; \phi \vdash \Theta : \Delta' \qquad \Delta; \phi \models R <: R'[\Theta][\theta]}{\Delta; \phi; \Psi; R \vdash \texttt{jmp}(v)}$$

$$\textbf{(ret)}\quad \frac{R(\texttt{csp}) = \forall\emptyset.\forall\phi'.\, R' :: S \qquad \phi \vdash \theta : \phi' \qquad \Delta; \phi \models R\{\texttt{csp} : S\} <: R'[\theta]}{\Delta; \phi; \Psi; R \vdash \texttt{ret}}$$

$$\textbf{(halt)}\quad \Delta; \phi; \Psi; R \vdash \texttt{halt}$$

$$\textbf{(seq)}\quad \frac{\Delta; \phi; \Psi; R \vdash ins \Rightarrow \phi'; R' \qquad \Delta; \phi'; \Psi; R' \vdash I}{\Delta; \phi; \Psi; R \vdash ins\ I}$$

$$\textbf{(body)}\quad \frac{\Delta; \phi; \Psi; R \vdash ins_1 \Rightarrow \phi_1; R_1 \ \cdots \ \Delta; \phi_{n-1}; \Psi; R_{n-1} \vdash ins_n \Rightarrow \phi'; R'}{\Delta; \phi; \Psi; R \vdash ins_1 \ \ldots \ ins_n \Rightarrow \phi'; R'}$$

Figure 3.11: Static semantics, instruction sequences

Note that, we let the loop counter runs from zero to the loop count, counting how many times the loop has been executed. This is different from what the abstract machine does and what the real hardware does, but it makes the type annotations we have to write in our programs somewhat nicer, and the difference is not observable in Featherweight DSP because the only instructions that can manipulate the do-stack is the `do` instruction and the `enddo`. See Chapter 4 for examples of such type annotations.

3.2.3 Typing of Instruction Sequences and Programs

Figure 3.11 shows the typing rules for instruction sequences and do-bodies. In the rule for jumps (**jmp**) we see the familiar pattern where substitutions and regfile subtyping are used for control transfer instructions. The rule for the return instruction (**ret**) is similar, except here we only need to find a substitution for the index variables, because we know from the (**call**) rule that the state types pushed on the call-stack do not bind any type variable. The rules (**call**) and (**ret**) are the only rules that manipulates the call-stack. The rule for the `halt` instruction simply states that it is correct to halt in any typing state.

The rules for instructions sequences (**seq**) and do-bodies (**body**) work by threading the index context and regfile type through the instructions comprising the sequence/do-body. These rules make the type system *control-flow sensitive*.

Figure 3.12 shows the typing rules for programs. That is, how to type check data values and instruction sequences in the initial store.

$$\textbf{(xarray)}\quad \frac{\Psi;\{\} \vdash v_1 : \tau_1 \quad \cdots \quad \Psi;\{\} \vdash v_n : \tau_n \qquad \emptyset;\{\} \models \tau_1 <: \tau \quad \cdots \quad \emptyset;\{\} \models \tau_n <: \tau}{\Psi \vdash \texttt{X:<}v_1,\ldots,v_n\texttt{>} : \tau\ \texttt{xarray}(n)}$$

$$\textbf{(code)}\quad \frac{\Delta;\phi;\Psi;R \vdash I}{\Psi \vdash I : \forall\Delta.\forall\phi.\, R}$$

$$\textbf{(prog)}\quad \frac{\begin{array}{c}\emptyset;\{\} \vdash_{\text{wf}} \tau_1 \quad \cdots \quad \emptyset;\{\} \vdash_{\text{wf}} \tau_n \\ \Psi = \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\ \Psi \vdash \mathit{mval}_1 : \tau_1 \quad \cdots \quad \Psi \vdash \mathit{mval}_n : \tau_n \\ \emptyset;\{\} \models \forall.\forall.\, R_{\mathit{init}} <: \tau_i\end{array}}{\vdash (\ell_i, \ell_1 : \mathit{mval}_1\ \ldots\ \ell_n : \mathit{mval}_n)}$$

Figure 3.12: Static semantics, programs

The most interesting thing to note about these rules is how parametric polymorphism is treated. The rule for type checking data values (**xarray**) and the rule for whole programs (**prog**) ensure that we cannot have any polymorphic data memory locations, because the values have to be wellformed in an empty type variable context and empty index context. Code labels, on the other hand, are allowed to introduce their own type and index variables without restrictions.

In the rule for whole programs (**prog**) when we check the that it is safe to the execution at label ℓ_i we use the special regfile type R_{init} that maps all registers to the nonsense type `junk`.

3.3 Properties of the Baseline Type System

The baseline type system ensures that if a program *prog* type checks $\vdash$ *prog*, then the abstract machine from Section 2.3.2 will not become stuck during execution of *prog*. That is, either the abstract machine will reach a terminal configurarion (i.e., a configuration where the current instruction sequence is `halt`) or it will run forever (i.e., there will always be a rewrite rule that match the current configuration).

This means that the type system ensures that fixed-point numbers are only added or multipied to fixed-point numbers, that integers are only added to other integers or to pointers into X or Y memory, that integers are only multiplied to other integers, that two pointers are neither multiplied nor added, that we only transfer control to instruction sequences and not to data values, that we cannot execute the `enddo` instruction unless we have branched out of a `do`-loop, a `ret` instruction unless an unmatched `call` instruction has been executed, and that we do not read or write outside the bounds of an array.

This section sketches the formalisation of these properties. The formalisation of the link between the operational semantics from Section 2.3.2 and the

baseline type system is messy because the operational semantics and the baseline type system have been developed for two different purposes, and not to match up nicely in a formal proof. As it is clear from the following, I have not carried out complete proofs for the theorems and lemmas I presents. This section just present a rough outline of the formalisation.

As the baseline type system is an adaptation of DTAL, we can also reuse the proof of type soundness for DTAL in [55] to prove type soundness for the baseline type system.

The proof follows the standard subject-reduction strategy [54]. Our main theorem is the theorem for type safety.

Theorem 3.1 (Type safety)
Let P be a program $(\ell_i,\ \ell_1 : mval_1 \quad \cdots \quad \ell_n : mval_n)$. If $\vdash P$ then P cannot become stuck during evaluation when we start the execution at the code label ℓ_i. □

Theorem 3.1 can be proved via the usual subject reduction and progress lemmas.

Lemma 3.3 (Subject reduction)
If $\vdash M$ and $M \blacktriangleright M'$ then $\vdash M'$. □

Lemma 3.4 (Progress)
If $\vdash M$ then either M is a terminal configuration (that is, the instruction sequence I is just `halt`*) or there exists a M' such that $M \blacktriangleright M'$.* □

In Lemma 3.3 and Lemma 3.4 we use the judgement $\vdash M$ of a well-typed machine configuration which has not been defined. Thus, we need to define what the judgement $\vdash M$ means.

Definition 3.4 (Well-typed Machine Configurations)
A machine configuration $M = (X, Y, C, \Gamma, S, D, I)$ is well-typed $\vdash M$ if we can find a typing context a store type Ψ and a regfile type R such that the following conditions are satisfied:

1. Ψ and R are wellformed: $\emptyset; \{\} \vdash_{\text{wf}} \Psi$ and $\emptyset; \{\} \vdash_{\text{wf}} R$.
2. The store type Ψ describe the X, Y, and code memory. That is, we need a judgement $\Psi \vdash X, Y, C$. This judgement is described in the following.
3. All the registers i Γ can be given the types in the the regfile type R: $\Psi; X; Y; C \vdash \Gamma(r) : R(r)$ for all $r \in \mathbf{dom}(\Gamma)$. Again, this judgement is describe in more detail in the following.
4. The call-stack can be given the type in $R(\texttt{csp})$: $\Psi \vdash S : R(\texttt{csp})$.
5. The do-stack corresponds to $R(\texttt{dsp})$ and that the current instruction sequence is well-typed: $\emptyset; \{\}; \Psi; R \vdash I, D$. □

In Definition 3.4 we have used some undefined helper judgements. These judgements are not defined because the baseline type system operates on *syntactic values* whereas the dynamic semantics operates on *dynamic values*. The

$$\textbf{(loc-xarray)}\quad \frac{\begin{array}{c} X(\ell) = (d_1, \ldots, d_n) \\ 0 \leq i \leq n \\ \Psi; X; Y; C \vdash d_1 : \tau_1 \quad \ldots \quad \Psi; X; Y; C \vdash d_n : \tau_n \\ \emptyset; \{\} \models \tau_1 <: \tau \quad \ldots \quad \emptyset; \{\} \models \tau_n <: \tau \end{array}}{\Psi; X; Y; C \vdash \langle \ell, i \rangle : \tau \ \texttt{xarray}(n-i)}$$

$$\textbf{(loc-code)}\quad \frac{\begin{array}{c} C(\ell) = I \\ \Delta; \phi; \Psi; R \vdash I \end{array}}{\Psi; X; Y; C \vdash \langle \ell, 0 \rangle : \forall \Delta. \forall \phi.\, R}$$

Figure 3.13: Static semantics, dynamic locations

most basic part we need to define is how to derive types for data values (d in Figure 2.8). That is, we need a judgement $\Psi; X; Y; C \vdash d : \tau$, notice that for this judgement we need both the dynamic and static store. Only the rules for locations are interesting. Figure 3.13 shows the typing rules for dynamic locations.

When we have the judgement for data values we can define a judgement for the whole store:

$$\frac{\begin{array}{ll} \Psi; X; Y; C \vdash X(\ell) : \Psi(\ell) & \text{for all } \ell \in \mathbf{dom}(X) \\ \Psi; X; Y; C \vdash Y(\ell) : \Psi(\ell) & \text{for all } \ell \in \mathbf{dom}(Y) \\ \Psi; X; Y; C \vdash C(\ell) : \Psi(\ell) & \text{for all } \ell \in \mathbf{dom}(C) \end{array}}{\Psi \vdash X, Y, C}$$

Similar we can make a judgement for the call stack S without problems. However, it is more troublesome to define the judgement for the do-stack D and for the current instruction sequence. The problems stems from the treatment of do-loops in the operational semantics, and in particular that the operational semantics is a small-step semantics whereas the typing rules for do-loops (**do**) and (**body**) in Figures 3.9 and 3.11 takes a more big-step view of do-loops. Thus, we need to define a judgement that takes both the do-stack and the current instruction sequence into account.

A property that we need to prove to show Lemma 3.4 is that the type system preserve consistent index contexts, or rather that consistent index contexts are preserved for reachable code.

Lemma 3.5 (Consistent Contexts are preserved)
Given Δ, ϕ, and R, where ϕ is consistent and R is well formed, and:

$$\Delta; \phi; \Psi; R \vdash ins \Rightarrow \phi'; R'$$

and control is transfered to the instruction following ins, then R' is wellformed and ϕ' is consistent.

□

Proof (Sketch) To prove Lemma 3.5 the only interesting rules to examinate are those from Figure 3.9 where the index context is changed. That is:

- Extending the context with a context bound by an existential quantifier the rule (**eelim**).
- The rules for conditional jump family exemplified by rule (**beq**).
- Subroutine call the rule (**call**).
- Do-loops the rule (**do**).

For all cases we can use Lemma 3.6 in the following, that says that if we have a consistent index context we cannot use it to find a substitution for an inconsistent index context.

For the rule (**eelim**) Lemma 3.6 works, because all existential quantifiers must have been introduced via the rule (**existr-sub**) from Figure 3.6 and here we see that the index context packaged by the existential quantifier must have been consistent to be packaged in the first place.

For the other rules the most interesting thing to note, is that we must allow inconsistent index context for unreachable code. For example, if we know that register r contains the integer 0, and we reach the instruction `beq` r, v, then we will create an inconsistent index context. This is not a problem, because the code is unreachable, nevertheless it is an uncommon property for a type system. ∎

Lemma 3.6 (Substitution preserve consistency)
If $\vdash_c \phi_1$ *and* $\phi_1 \vdash \theta : \phi_2$ *then* $\vdash_c \phi_2$. □

Proof Follow straight from the definition of consistency (Definition 3.3) and the substitution judgement in Figure 3.4. ∎

3.4 Shortcomings of the Baseline Type System

This section describes two problems I have found with the DTAL-like baseline type system presented in Section 3.2. These problems have been identified by trying to give type annotations to handwritten DSP assembler programs. Sections 3.5 and 3.6 present extentions to the baseline type system to overcome these shortcomings.

3.4.1 Invariance of the array type constructors

The invariance of the type constructors `xarray` and `yarray` (see Section 3.1.8) can hinder us for giving a precise type to a procedure that manipulates an array.

As an example, consider the procedure `fill_zero` in Figure 3.14. This procedure takes an array (in X memory) and initialises the array with zeros. This procedure is an example of an important class of procedures. We only have statically allocated memory that we have to manage explicitly; we cannot rely on a runtime system to initialise memory for us. Our first stab at a type for

```
1   fill_zero:
2       x0 = 0
3       do (i11) {
4          xmem[i0] = x0; i0 += 1
5       }
6       ret
```

Figure 3.14: Initialisation of array.

`fill_zero` could be:

$$\begin{aligned}\forall s.\forall\{n : \texttt{int} \mid n > 0\}.[\ & \texttt{i0} : \texttt{junk xarray}(n),\\ & \texttt{i11} : \texttt{int}(n),\\ & \texttt{csp} : [\ \texttt{i0} : \texttt{int}(0)\ \texttt{xarray}(n),\\ & \qquad \texttt{i11} : \texttt{int}(n),\\ & \qquad \texttt{x0} : \texttt{junk},\\ & \qquad \texttt{csp} : s] :: s]\end{aligned}$$

This type says that when we call `fill_zero` then the register `i0` must contain (a pointer to) an array in X memory of size n, the register `i11` must contain the integer n, and the top of the call-stack must be a code-pointer that expects the register `i0` to contain an array of size n where all the elements have the value 0, and the register `x0` to contain some arbitrary value. But this type is wrong for two reasons: First, while the rule (**write**) in Figure 3.8 allows us to write integers (with value 0) to a `junk` array, there is no way to transform a `junk` array into an integer array, even if we know operationally that all the elements of the array are integers (with the value 0). Second, the procedure `fill_zero` in Figure 3.14 increments the register `i0` n times, thus `i0` will contain an array of size zero. This could be fixed by saving the value of `i0` in the beginning of `fill_zero`, but that would still not fix the first problem. Hence, the best type we end up with for `fill_zero` is:

$$\begin{aligned}\forall s.\forall\{n : \texttt{int} \mid n > 0\}.[\ & \texttt{i0} : \texttt{int xarray}(n),\\ & \texttt{i11} : \texttt{int}(n),\\ & \texttt{csp} : [\ \texttt{i0} : \texttt{int xarray}(0),\\ & \qquad \texttt{i11} : \texttt{int}(n),\\ & \qquad \texttt{x0} : \texttt{junk},\\ & \qquad \texttt{csp} : s] :: s]\end{aligned}$$

which is not satisfying because this type does not capture the main functionality of `fill_zero`. (In this type I have even cheated a bit and use `int` as shorthand for $\exists\{k : \texttt{int}\}.\texttt{int}(k)$.)

```
1  vecpmult_prefetch:
2      x0  = xmem[i0]; i0+=1
3      y0  = ymem[i4]; i4+=1
4      do (i7) {
5         a0=x0*y0; x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
6         xmem[i1] = a0; i1+=1
7      }
8      ret
```

Figure 3.15: Pointwise vector multiplication with prefetch.

Another consequence of the invariance of the array type constructors is that we must be careful not to be too specific when we give a type for an array. For example, if we have an array with the type:

$$\texttt{int}(0)\ \texttt{xarray}(n)$$

then we are only allowed to write integers with the value zero into this array.

In Section 3.6 I present some modifications to the baseline type system so that `fill_zero` can be typed, and in Section 4.1.2 I show the type annotations for `fill_zero` using this modified type system.

3.4.2 Prefetching from Memory

A common idiom found in loops that traverse an array in sequence is to prefetch data from memory so that the data is ready in registers when needed for calculations. Using this idiom together with composite instructions it is often possible to reduce the number of instructions (not small instructions) in a loop.

Figure 3.15 shows a procedure that performs pointwise vector multiplication using this idiom. Compare this code to the, less efficient, code in Figure 2.2 on page 18. The type we want to assign to `vecpmult_prefetch` is:

$$\begin{aligned}\forall s.\forall\{n : \texttt{int} \mid n > 0\}.[\ &\texttt{i0} : \texttt{fix xarray}(n),\\ &\texttt{i1} : \texttt{fix xarray}(n),\\ &\texttt{i4} : \texttt{fix yarray}(n),\\ &\texttt{i7} : \texttt{int}(n),\\ &\texttt{csp} : [\ \texttt{x0} : \texttt{junk}, \texttt{y0} : \texttt{junk}, \texttt{a0} : \texttt{junk},\\ &\qquad \texttt{i0} : \texttt{fix xarray}(0),\\ &\qquad \texttt{i1} : \texttt{fix xarray}(0),\\ &\qquad \texttt{i4} : \texttt{fix yarray}(0),\\ &\qquad \texttt{i7} : \texttt{int}(n),\\ &\qquad \texttt{csp} : s] :: s]\end{aligned}$$

There is nothing wrong with this type if we use the code from Figure 2.2 which does not prefetch data from memory, but in Figure 3.15 the array pointers in the registers `i0` and `i4` are incremented $n + 1$ times. Thus, the length of the arrays in `i0` and `i4` in the return type should be -1 and not 0. This can be fixed, but what cannot be fixed is that in the last round of the `do`-loop the values of `x0` and `y0` (line 5) are read outside the bounds of the arrays in `i0` and `i4`, and thus the program does not type check. In the last round of the `do`-loop the size of the arrays in `i0` and `i4` will be zero, thus the (**read**) rule from Figure 3.8 cannot be used since it states that e (the length of the array) must be greater than zero.

In the following section I show how we can work around this problem, and Section 4.1.1 and Section 4.1.2 give type annotated versions of vector multiplication with and without the prefetch idiom.

3.5 Extension 1: Out of Bounds Memory Reads

As pointed out in Section 3.4.2 the prefetch example from Figure 3.15 is not typable using the baseline type system presented in Section 3.2. There are two reasons why the prefetch example cannot be typed (the second is a consequence of the first):

1. In the last round of the `do`-loop the code reads from outside the bounds of the arrays being multiplied (the arrays pointed to by `i0` and `i4`). But values read are never used in a computation. The obvious solution to fix this problem is to allow reads from out of bounds memory, but give the type `junk` to the data read (there are no instructions that works on `junk`, thus this is safe).

2. If we modify the rules to allow out of bounds memory reads as described in the first point then we face the problem that in the last round of the `do`-loop in the prefetch example the register file, R, will have a different *shape* than the other rounds. To be more specific: after all the rounds, but the last, the registers `x0` and `y0` have type `fix`; whereas after the last round the registers `x0` and `y0` have type `junk`.

Thus, to be able to type the prefetch example we need to extend the rules from Figure 3.8 with a rule to allow out of bounds memory reads. And we need to refine the rule for `do`-loops from Figure 3.9 to allow that in the last round of a loop the regfile type can be different from all the other rounds. Figure 3.16 shows these rules. Like in Figure 3.8 I only show a rule for X memory, as the rule for Y memory is similar.

You may wonder why it is necessary to change the type rule for `do`-loops to allow out-of-bounds memory reads since `do`-loops do not have anything to do with reading from memory. The reason for this is that with the (**read-oob**) rule we depart in a significant way from DTAL. Namely, we allow the index context to determine the syntactic structure of types in the conclusion of a typing rule.

(**read-oob**)
$$\frac{\Psi; R \vdash r_2 : \tau\ \texttt{xarray}(e) \qquad \phi \models e \leq 0}{\Delta; \phi; \Psi; R \vdash r_1\ \texttt{= xmem[}r_2\texttt{]} \Rightarrow R\{r_1 : \texttt{junk}\}}$$

(**do-oob**)
$$\frac{\begin{array}{c}\Psi; R_1 \vdash v : \texttt{int}(e) \qquad \phi_1 \models e > 0 \\ R_1(\texttt{dsp}) = S \\ k_1 \in \mathbf{dom}(\phi_2) \qquad \phi_2 \models 0 \leq k_1 < e \\ R_2(\texttt{dsp}) = \texttt{int}(k_1) :: S \\ \phi_1 \vdash \theta_1 : \phi_2 \\ \Delta; \phi_1 \models R_1\{\texttt{dsp} : \texttt{int}(0) :: S\} <: R_2[\theta_1] \\ \Delta; \phi_2 \wedge 0 \leq k_1 < e - 1; \Psi; R_2 \vdash B \Rightarrow \phi_3; R_3 \\ \phi_3' = \phi_3 \wedge \{k_2 : \texttt{int} \mid k_1 < e - 2 \wedge k_2 = k_1 + 1\} \qquad k_2 \notin \mathbf{dom}(\phi_3) \\ \phi_3' \vdash \theta_2 : \phi_2 \\ \Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: S\} <: R_2[\theta_2] \\ \Delta; \phi_2 \wedge k_1 = e - 1; \Psi; R_2 \vdash B \Rightarrow \phi_4; R_4\{\texttt{dsp} : \texttt{int}(k_1) :: S\}\end{array}}{\Delta; \phi_1; \Psi; R_1 \vdash \texttt{do(}v\texttt{) \{}B\texttt{\}} \Rightarrow \phi_4; R_4\{\texttt{dsp} : S\}}$$

Figure 3.16: Rule for out of bounds memory reads and refined rule for do-loops.

Thus, we cannot just erase all the dependent types from our derivation trees and still get a valid derivation tree. This means that if we extend the baseline type system we can make more programs typable because we have more precise types.

Instead of modifying the (**do**) rule we could introduce some kind of sum types and make the result of load instruction be a sum type where one summand would be junk if we read outside the bounds of an array. In Section 6.1.2 I go into more details about a sum-type extension.

The two extended typing rules in Figure 3.16 have two drawbacks:

- They make it more difficult to give precise type error messages when there is an out-of-bounds error, because we now allow values to be read from outside an array, even if that is not what the programmer intended. Instead a type error will occur when the values read outside an array are used in a calculation, and these two places (the read and the use) may be far from each other in the program text.

- The rule (**do-oob**) requires that the body of a loop, B, is checked twice. Thus, if we have nested loops the innermost body is checked an exponential number of times proportional to the depth of the nesting.

3.6 Extension 2: Pointer Arithmetics and Aggregate Types

In Section 3.4.1 we saw that if we use the baseline type system presented in Section 3.2 we cannot give a type that captures the main functionality of the typical

store types	Ψ	::=	$\{\rho_1 : \Xi_1, \ldots, \rho_m : \Xi_m\}_X *$ $\{\rho'_1 : \Xi'_1, \ldots, \rho'_n : \Xi'_n\}_Y *$ $\{\ell_1 : \sigma_1, \ldots, \ell_k : \sigma_k\}$
state types	σ	::=	$\forall\Delta.\forall\phi.\,(\Psi, R)$
locations	ρ	::=	$\ell \mid \eta$
type variable contexts	Δ	::=	$\emptyset \mid \Delta, \omega \mid \Delta, \alpha \mid \Delta, \eta$
types	τ	::=	$\alpha \mid \sigma \mid \exists\phi.\tau \mid \texttt{junk} \mid \texttt{int}(e) \mid \texttt{fix} \mid$ $\texttt{xptr}(\rho, e) \mid \texttt{yptr}(\rho, e)$
aggregate types	Ξ	::=	$\tau[e] \mid \Xi_1 @ \Xi_2$
location variables	η		

Figure 3.17: Type syntax for Featherweight DSP extended with locations and aggregate types.

function `fill_zero` in Figure 3.14. This section presents modified version of the baseline type system. The modification consist of two novel typing constructs: a combination of alias types and index types that allows us to track alias information and pointer arithmetic, and *aggregate types* that allow us to handle non-homogeneous arrays and have different views on a block of memory.

The reason we cannot give a satisfactory type for `fill_zero` is that the baseline type system handles alias information in such a heavy-handed way: by enforcing the type invariance principle. That is, all memory locations are required to be invariant in their type, as are the array type constructors. The assumption that memory locations are invariant in their type makes it sound to ignore alias information which is sometimes a complication desirable to get rid of. But for low-level languages such as assembler, alias information can be important when we want to give a more precise type for functions like `fill_zero`, for instance. The important functionality of `fill_zero` is how it modifies the store.

Let us take a look at the procedure `fill_zero` again to figure out what is needed to type check this procedure precisely. As it happens, it is not enough just to track alias information. We must also handle that, within the `do`-loop, the array that `fill_zero` manipulates is only partially initialised. That is, the first part of the array has been initialised but the last part still contains nonsense values. Thus, we need to extend the baseline type system so that it can handles: alias information, non-homogeneous arrays (that is, the elements of an array can have different types), and pointers into these non-homogeneous arrays. Henceforth I use the term *aggregate objects* to denote non-homogeneous arrays (borrowing a term from the C standard [27]).

Figure 3.17 shows the modified parts of the syntax for type expressions. Compared to the syntax in Figure 3.1, the modifications are:

- locations, ρ, are now either a concrete label, ℓ, or a location variable, η;
- two new type constructors $\texttt{xptr}(\rho, e)$ and $\texttt{yptr}(\rho, e)$. A value with type $\texttt{xptr}(\rho, e)$ denotes a pointer to the address $\rho + e$ in X memory;

- state types, σ, are extended with a store type component;
- a new form of types called segment types, $\tau[e]$, to denote sequences of elements of the same type. That is, the segment $\tau[e]$ denotes e elements of type τ. The elements are consecutive in memory. We say that e is the size of the segment.
- aggregate types, Ξ, which are sequences of segments;
- no array types, because we use aggregate types instead;
- store types, Ψ, are spilt into three mappings: one for locations in X memory, one for locations in Y memory, and one for locations in code memory.

The first three modifications are used for tracking alias information in the style of Walker [53], the next two modifications are for handling aggregate objects, and the last two modifications are just to simplify some of the typing rules.

Aggregate types can on one hand be seen as a generalisation of product types. For example, to model a tuple with three integers we can freely use one of the following aggregate types:

$$\texttt{int}[1]\texttt{@int}[1]\texttt{@int}[1]$$
$$\texttt{int}[1]\texttt{@int}[2]$$
$$\texttt{int}[2]\texttt{@int}[1]$$
$$\texttt{int}[3]$$

which are all equivalent. If we want to write the type of an array of fixed-point numbers that starts with a header specifying the length of the array as an integer, we can use the aggregate type:

$$\texttt{int}(n)[1]\texttt{@fix}[n]$$

On the other hand, aggregate types are not as "first class" as ordinary tuples. While tuples can usually be nested; we do not allow the formation of segments where elements types are aggregate types themselves. The reason for this is that it would take us outside Presburger arithmetic. I go into more detail about nested aggregate types in Section 6.1.2.

3.6.1 Equality for Pointer Types and Aggregate Types

Figure 3.18 extends the type equality defined in Figure 3.5 to pointer types and aggregate types. The rules are not syntax directed at all, instead it should be clear that they define an equivalence relation. What is perhaps more interesting, is that these rules are defined so that aggregate types form a *monoid* with the append operator `@` as composition and any segment of size zero as the identity (or unit). That is, the following properties are satified:

- It has left and right *units*. Any segment of size zero is a unit. This follows directly from the (**aggr-zerol-eq**) and the (**aggr-zeror-eq**) rules.

$$\textbf{(ptr-var-eq)}\quad \frac{\eta_1 \in \Delta \qquad \eta_2 \in \Delta \qquad \eta_1 = \eta_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \texttt{xptr}(\eta_1, e_1) \equiv \texttt{xptr}(\eta_2, e_2)}$$

$$\textbf{(ptr-loc-eq)}\quad \frac{\ell_1 = \ell_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \texttt{xptr}(\ell_1, e_1) \equiv \texttt{xptr}(\ell_2, e_2)}$$

$$\textbf{(aggr-seg-eq)}\quad \frac{\Delta; \phi \models \tau_1 \equiv \tau_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \tau_1[e_1] \equiv \tau_2[e_2]}$$

$$\textbf{(aggr-zerol-eq)}\quad \frac{\phi \models e = 0}{\Delta; \phi \models \tau[e] \texttt{@} \Xi \equiv \Xi}$$

$$\textbf{(aggr-zeror-eq)}\quad \frac{\phi \models e = 0}{\Delta; \phi \models \Xi \texttt{@} \tau[e] \equiv \Xi}$$

$$\textbf{(aggr-split-eq)}\quad \frac{\Delta; \phi \models \tau \equiv \tau_1 \qquad \Delta; \phi \models \tau \equiv \tau_2 \qquad \phi \models e = e_1 + e_2 \qquad \phi \models e_1 \geq 0 \qquad \phi \models e_2 \geq 0}{\Delta; \phi \models \tau[e] \equiv \tau_1[e_1] \texttt{@} \tau_2[e_2]}$$

$$\textbf{(aggr-assoc-eq)}\quad \Delta; \phi \models \Xi_1 \texttt{@} (\Xi_2 \texttt{@} \Xi_3) \equiv (\Xi_1 \texttt{@} \Xi_2) \texttt{@} \Xi_3$$

$$\textbf{(aggr-cong-eq)}\quad \frac{\Delta; \phi \models \Xi_1 \equiv \Xi_1' \qquad \Delta; \phi \models \Xi_2 \equiv \Xi_2'}{\Delta; \phi \models \Xi_1 \texttt{@} \Xi_2 \equiv \Xi_1' \texttt{@} \Xi_2'}$$

$$\textbf{(aggr-trans-eq)}\quad \frac{\Delta; \phi \models \Xi_1 \equiv \Xi_2 \qquad \Delta; \phi \models \Xi_2 \equiv \Xi_3}{\Delta; \phi \models \Xi_1 \equiv \Xi_3}$$

Figure 3.18: Equality for pointer and aggregate types

- It is *associative*. This follows directly from the (**aggr-assoc-eq**) rule.

Because `@` forms a monoid we are justified to view an aggregate type as a *sequence* of segments. In the rest of this thesis I shall write aggregate types as sequences without further comment.

Aggregate types defined solely by the syntactic grammar and restricted only by the wellformedness constraint that all type variables and index variables must be bound in a typing context are too flexible, it allows us to write nonsensical segments such as $\tau[-1]$. The wellformedness rules for aggregate types must exclude segments with a negative size. Figure 3.19 shows the rules for well-formed pointer types and aggregate types. But even well-formed aggregate types can be too unwieldy sometimes, the notion of *normalised* aggregate types can be convenient.

$$\frac{\eta \in \Delta \qquad \phi \vdash_{\text{wf}} e}{\Delta; \phi \vdash_{\text{wf}} \texttt{xptr}(\eta, e)} \qquad \frac{\phi \vdash_{\text{wf}} e}{\Delta; \phi \vdash_{\text{wf}} \texttt{xptr}(l, e)}$$

$$\frac{\Delta; \phi \vdash_{\text{wf}} \tau \qquad \phi \models 0 \leq e}{\Delta; \phi \vdash_{\text{wf}} \tau[e]} \qquad \frac{\Delta; \phi \vdash_{\text{wf}} \Xi_1 \qquad \Delta; \phi \vdash_{\text{wf}} \Xi_2}{\Delta; \phi \vdash_{\text{wf}} \Xi_1 \texttt{@} \Xi_2}$$

Figure 3.19: Well-formed pointer types and aggregate types

Definition 3.5 (Normalised aggregate types)
An aggregate type $\tau_1[e_1]\ \texttt{@} \cdots \texttt{@}\ \tau_n[e_n]$ is *normalised* with respect to a type variable context Δ and an index context ϕ if and only if:

1. $\phi \models 0 < e_i$ for all e_i.
2. All adjacent segments $\tau_i[e_i]\ \texttt{@}\ \tau_{i+1}[e_{i+1}]$ have distinct element types. That is, $\Delta; \phi \models \tau_i \equiv \tau_{i+1}$ does not hold. □

3.6.2 Subtyping for Pointer Types and Aggregate Types

Similar to how we defined equality, in the previous section we need to extend the rules for subtyping, and we need to define a new typing rule for state types, because state types now include a store type. Figure 3.20 shows the new rules for subtyping of pointer types, aggregate types, state types, and store types.

Again, the rules for aggregate type are not syntax directed at all, instead it should be clear that they define a partial ordering. The rules for pointer types and aggregate types are straightforward and should not cause any surprises.

In the rule for state types (**state-xp-sub**) we need a substitution on location variables, but there is a slight twist on which substitutions are allowed. This is discussed in the following section. Other than that, the rule (**state-xp-sub**) is the obvious extension of the rule (**state-sub**) from Figure 3.6.

The rule for store types (**store-sub**) is similar to the rule for regfile types (**regs-sub**) in Figure 3.6. The only noteworthy part is that we do not have to go through all three components of the store types only the parts for X memory and Y memory. The code memory part does not change during type checking, this will be enforced by the rule for whole programs in Section 3.6.4.

3.6.3 Substitutions for Location Variables

For the judgement for subtyping of state types we need to define substitution for pointer variables. Thus, we extend type and stack variable substitutions, Θ, from Section 3.1.6:

$$\Theta \quad ::= \quad [] \mid \Theta[\alpha \mapsto \tau] \mid \Theta[\omega \mapsto S] \mid \Theta[\eta \mapsto \rho]$$

(**ptr-var-sub**)
$$\frac{\eta_1 \in \Delta \qquad \eta_2 \in \Delta \qquad \eta_1 = \eta_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \texttt{xptr}(\eta_1, e_1) <: \texttt{xptr}(\eta_2, e_2)}$$

(**ptr-loc-sub**)
$$\frac{\ell_1 = \ell_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \texttt{xptr}(\ell_1, e_1) <: \texttt{xptr}(\ell_2, e_2)}$$

(**aggr-seg-sub**)
$$\frac{\Delta; \phi \models \tau_1 <: \tau_2 \qquad \phi \models e_1 = e_2}{\Delta; \phi \models \tau_1[e_1] <: \tau_2[e_2]}$$

(**aggr-join-sub**)
$$\frac{\phi \models e = e_1 + e_2 \qquad \phi \models e_1 \geq 0 \qquad \phi \models e_2 \geq 0 \qquad \Delta; \phi \models \tau_1 <: \tau \qquad \Delta; \phi \models \tau_2 <: \tau}{\Delta; \phi \models \tau_1[e_1] @ \tau_2[e_2] <: \tau[e]}$$

(**aggr-split-sub**)
$$\frac{\phi \models e = e_1 + e_2 \qquad \phi \models e_1 \geq 0 \qquad \phi \models e_2 \geq 0 \qquad \Delta; \phi \models \tau <: \tau_1 \qquad \Delta; \phi \models \tau <: \tau_2}{\Delta; \phi \models \tau[e] <: \tau_1[e_1] @ \tau_2[e_2]}$$

(**aggr-refl-sub**)
$$\frac{\Delta; \phi \models \Xi_1 \equiv \Xi_2}{\Delta; \phi \models \Xi_1 <: \Xi_2}$$

(**aggr-cong-sub**)
$$\frac{\Delta; \phi \models \Xi_1 <: \Xi_1' \qquad \Delta; \phi \models \Xi_2 <: \Xi_2'}{\Delta; \phi \models \Xi_1 @ \Xi_2 <: \Xi_1' @ \Xi_2'}$$

(**aggr-trans-eq**)
$$\frac{\Delta; \phi \models \Xi_1 <: \Xi_2 \qquad \Delta; \phi \models \Xi_2 <: \Xi_3}{\Delta; \phi \models \Xi_1 <: \Xi_3}$$

(**state-xp-sub**)
$$\frac{\Delta \cap \Delta_2 = \emptyset \qquad \mathbf{dom}(\phi) \cap \mathbf{dom}(\phi_2) = \emptyset \qquad \phi_2 \vdash \theta : \phi_1 \qquad \Delta_2; \phi_2 \vdash \Theta : \Delta_1 \qquad \Delta, \Delta_2; \phi \wedge \phi_2 \models R_2 <: R_1[\Theta][\theta] \qquad \Delta, \Delta_2; \phi \wedge \phi_2 \models \Psi_2 <: \Psi_1[\Theta][\theta]}{\Delta; \phi \models \forall \Delta_1. \forall \phi_1. (\Psi_1, R_1) <: \forall \Delta_2. \forall \phi_2. (\Psi_2, R_2)}$$

(**store-sub**)
$$\frac{\Delta; \phi \models \Psi_X(\rho) <: \Psi_X'(\rho) \quad \text{for all } \rho \text{ in } \Psi_X' \qquad \Delta; \phi \models \Psi_Y(\rho) <: \Psi_Y'(\rho) \quad \text{for all } \rho \text{ in } \Psi_Y'}{\Delta; \phi \models \Psi <: \Psi'}$$

Figure 3.20: Subtyping for pointer types, aggregate types, state types, and store types

But we have to be careful with these substitutions or we will introduce type unsoundness. In Walker and Morrisett's approach to alias information if two location variables are different then they must be *guaranteed* to point to different locations. In other words, we have the extra constraint that sharing or aliasing must not be introduced by substitution. This means that we extend the rules in Figure 3.4 with the two rules:

$$\frac{l \notin \mathbf{rng}(\Theta) \quad \Delta; \phi \vdash \Theta : \Delta'}{\Delta; \phi \vdash \Theta[\eta \mapsto \ell] : \Delta', \eta} \qquad \frac{\eta_2 \in \Delta \quad \eta_2 \notin \mathbf{rng}(\Theta) \quad \Delta; \phi \vdash \Theta : \Delta'}{\Delta; \phi \vdash \Theta[\eta_1 \mapsto \eta_2] : \Delta', \eta_1}$$

3.6.4 Instructions, Instruction Sequences, and Programs

The major change we have to make relative to the rules for instructions and instruction sequences in baseline type system is that state types now have a store component and that we have to thread a store type through the typing rules. That is, for small instructions the typing judgement is changed from

$$\Delta; \phi; \Psi; R \vdash \mathit{sins} \Rightarrow R'$$

to

$$\Delta; \phi; \Psi; R \vdash \mathit{sins} \Rightarrow \Psi'; R'$$

For instructions, the typing judgement is similarly changed from

$$\Delta; \phi; \Psi; R \vdash \mathit{ins} \Rightarrow \phi'; R'$$

to

$$\Delta; \phi; \Psi; R \vdash \mathit{ins} \Rightarrow \phi'; \Psi'; R'$$

This change is pervasive, but for most of the rules, the required modifications are straightforward and follow a common pattern. Thus, in this section I only show the interesting rules or describe common patterns.

The rules that require nontrivial changes are the rules for the small instructions that manipulate the store and manipulate pointers. Figure 3.21 shows the new rules for reading from memory, writing to memory, and a sample rule for how to increment a pointer (to X memory). Like in Section 3.2.2 I do not list all the rules for doing pointer arithmetic because they are all similar. But the restriction that pointers only can be incremented is now removed.

The rule for incrementing a pointer (**incr-pa**) is pleasingly simple. From the rule it is obvious that incrementing a pointer does not modify the store (because Ψ is the same on both the left-hand side and the right-hand side of the judgement in the conclusion). And the machinery for handling pointer arithmetic is now almost identical to the machinery for handling integer arithmetic. The only difference is that pointers are oriented around an "offset-location" ρ.

$$\textbf{(read-pa)}\quad \frac{\begin{array}{c} R(r_2) = \texttt{xptr}(\rho, e) \\ \Delta; \phi \models \Psi_X(\rho) \equiv \tau_1[e_1]\texttt{@}\cdots\texttt{@}\tau_n[e_n] \\ \phi \models 0 \leq e < e_1 + \cdots + e_n \\ \phi \models 0 \leq e - (e_1 + \cdots + e_{i-1}) < e_i \end{array}}{\Delta; \phi; \Psi; R \vdash r_1 \texttt{ = xmem[}r_2\texttt{]} \Rightarrow \{\}; [r_1 : \tau_i]}$$

$$\textbf{(write-pa)}\quad \frac{\begin{array}{c} R(r_1) = \texttt{xptr}(\rho, e) \qquad R(r_2) = \tau \\ \Delta; \phi \models \Psi_X(\rho) \equiv \tau_1[e_1]\texttt{@}\cdots\texttt{@}\tau_n[e_n] \\ \phi \models 0 \leq e < e_1 + \cdots + e_n \\ \mathit{rest} = e - (e_1 + \cdots + e_{i-1}) \\ \phi \models 0 \leq \mathit{rest} < e_i \\ \Xi = \tau_1[e_1]\texttt{@}\cdots\texttt{@}\,\tau_i[\mathit{rest}]\,\texttt{@}\,\tau[1]\,\texttt{@}\,\tau_i[e_i - \mathit{rest} - 1]\,\texttt{@}\cdots\texttt{@}\,\tau_n[e_n] \end{array}}{\Delta; \phi; \Psi; R \vdash \texttt{xmem[}r_1\texttt{]} \texttt{ = } r_2 \Rightarrow \{\rho \mapsto \Xi\}_X; []}$$

$$\textbf{(incr-pa)}\quad \frac{R(r) = \texttt{xptr}(\rho, e)}{\Delta; \phi; \Psi; R \vdash r \texttt{ += } c \Rightarrow \{\}; [r : \texttt{xptr}(\rho, e + c)]}$$

Figure 3.21: Type rules for aliasing and pointer arithmetic

The rules for loading and storing to memory are, unsurprisingly, more involved. In the rule for loading from memory (**read-pa**) we first check that we have a pointer into X memory, $\texttt{xptr}(e, \rho)$, and that this pointer has not been incremented or decremented so much that it does not point into the aggregate object at ρ. Next, to find the type of the the value we load from memory we need to find the appropriate segment $\tau_i[e_i]$ that contains element number e of the aggregate object. The rule for storing to memory (**write-pa**) is similar, but when we have found the appropriate segment $\tau_i[e_i]$ that contains element number e we need to split this segment into three segments: the part before the value we are storing, $\tau_i[e - (e_1 + \cdots + e_{i-1})]$; the value we are storing, $\tau[1]$; and the part after the value we are storing, $\tau_i[e_i - \mathit{rest} - 1]$. It is important to note that these three segments all are well-formed, that is they have a size that is non-negative, but the first and the last segments can have size zero (if we are updating the first or the last element of the segment).

In the typing rules for instructions and instruction sequences we have to change the rules so that not only the index context and a regfile type but also a store type is threaded through the rules. For composite instructions we have to change Merge so that it can merge store types as well as regfile types (since small instructions now return both a regfile type and a store type). For control transfer instructions we not only have to check that the register files are compatible, the store types have to be compatible too. For example, the typing rule for

(**aggr-pa**)
$$\frac{\Psi;\{\} \vdash v_1 : \tau_1 \quad \cdots \quad \Psi;\{\} \vdash v_n : \tau_n}{\Psi \vdash \texttt{<}v_1, \ldots, v_n\texttt{>} : \tau_1[1]\texttt{@}\cdots\texttt{@}\tau_n[1]}$$

(**code-pa**)
$$\frac{\begin{array}{c}\Psi = \{\rho_1 : \Xi_1, \ldots, \rho_m : \Xi_m\}_X * \\ \{\rho'_1 : \Xi'_1, \ldots, \rho'_n : \Xi'_n\}_Y * \\ \{\ell_1 : \sigma_1, \ldots, \ell_k : \sigma_k\} \\ \Delta;\phi \vdash_{\text{wf}} \Psi \qquad \Delta;\phi;\Psi;R \vdash I\end{array}}{\{\ell_1 : \sigma_1, \ldots, \ell_k : \sigma_k\} \vdash I : \forall\Delta.\forall\phi.\,(\Psi, R)}$$

(**prog-pa**)
$$\frac{\begin{array}{c}\emptyset;\{\} \vdash_{\text{wf}} \Xi_1 \quad \cdots \quad \emptyset;\{\} \vdash_{\text{wf}} \Xi_m \\ \emptyset;\{\} \vdash_{\text{wf}} \Xi'_1 \quad \cdots \quad \emptyset;\{\} \vdash_{\text{wf}} \Xi'_n \\ \Psi = \{\ell_{x1} : \Xi_1, \ldots, \ell_{xm} : \Xi_m\}_X * \\ \{\ell_{y1} : \Xi'_1, \ldots, \ell_{yn} : \Xi'_n\}_Y * \\ \{\ell_{c1} : \sigma_1, \ldots, \ell_{ck} : \sigma_k\} \\ \Psi \vdash \mathit{dval}_1 : \Xi_1 \quad \cdots \quad \Psi \vdash \mathit{dval}_m : \Xi_m \\ \Psi \vdash \mathit{dval}'_1 : \Xi'_1 \quad \cdots \quad \Psi \vdash \mathit{dval}'_n : \Xi'_n \\ \{\ell_{c1} : \sigma_1, \ldots, \ell_{ck} : \sigma_k\} \vdash \ell_{c1} : \sigma_1 \quad \cdots \quad \{\ell_{c1} : \sigma_1, \ldots, \ell_{ck} : \sigma_k\} \vdash \ell_{ck} : \sigma_k \\ \emptyset;\{\} \models \forall.\forall.\,(\Psi, R_{init}) <: \sigma_i\end{array}}{\begin{array}{l}\vdash (\ell_{ci},\ \ell_{x1} : \texttt{X:}\mathit{dval}_1 \ldots \ell_{xm} : \texttt{X:}\mathit{dval}_m \\ \qquad \ell_{y1} : \texttt{Y:}\mathit{dval}'_1 \ldots \ell_{yn} : \texttt{Y:}\mathit{dval}'_n \\ \qquad \ell_{c1} : I_1 \ldots \ell_{ck} : I_k)\end{array}}$$

Figure 3.22: Modified typing rules for programs and memory values

unconditional jumps becomes:

(**jmp-pa**)
$$\frac{\begin{array}{c}\Psi;R \vdash v :\ \forall\Delta'.\forall\phi'.\,(\Psi', R') \\ \phi \vdash \theta : \phi' \qquad \Delta;\phi \vdash \Theta : \Delta' \\ \Delta;\phi \models R <: R'[\Theta][\theta] \\ \Delta;\phi \models \Psi <: \Psi'[\Theta][\theta]\end{array}}{\Delta;\phi;\Psi;R \vdash \texttt{jmp(}v\texttt{)}}$$

and the rules for conditional branches, the `call` and `ret` instructions, and `do`-loops are changed in similar ways.

Figure 3.22 presents the new typing rules for programs. Here the typing rules for arrays have been replaced with rules for aggregate objects (**xaggr-pa**), the rules are similar except that for aggregate objects the elements do not all have to have the same type. The rule for code instructions (**code-pa**) has been changed so that the only assumptions on which procedures have to agree globally is the type of code locations. The rule for whole programs (**prog-pa**) looks complicated because store types have been split into three parts, but it is not much more complicated than the rule (**prog**) from Figure 3.12 on page 51; it is just more verbose.

The rules in Figure 3.22 are more complicated than the rules in Figure 3.12 for three reasons: First, the types of instruction sequences and data values are now different syntactic classes of types. Second, we don't split aggregate types into X and Y variants like we did for arrays in the baseline type system, instead this splitting is done in the store type, Ψ, in the rule (**prog-pa**). Finally, since we now thread the store type during checking, we are now allowed to put more requirements on the store type required for a given procedure. The last reason is the feature we are after, because it makes it possible to specify that the store must be of a certain type before a given procedure can be called. A curious detail of the rule (**code-pa**) is that it allow that some procedures might refer to constant locations in X or Y memory that do not exists. This is allowed as long as the procedures are not reachable from the start procedure, that is, these procedures must be dead code.

3.7 Summary

In this chapter we have seen that it is possible to adapt DTAL to Featherweight DSP, and in the next chapter we shall see that with this baseline type system we are able to express many invariants for handwritten DSP assembler programs. The main work of the adaptation of DTAL to Featherweight DSP has been to figure out how to handle the unusual features of a DSP assembler language like composite instructions and special loop syntax.

Then, guided by real-life examples, I described two shortcomings stemming from some of the DTAL design decisions: inability to handle the common idiom of prefetching memory and overly simplified handling of alias information. These design decisions might be valid and right for the original scope of DTAL (a typed low-level intermediate target language not intended for human use) where we can rely on a runtime system with a garbage collector that will take care of part of the memory management. But for handwritten DSP assembler these decisions must be revised.

Finally, I presented two orthogonal extensions to the baseline type system. The first extension is to allow out-of-bounds memory reads. The second extension is based on two major modifications of the baseline type system:

- The original DTAL enforces the type invariance principle of memory locations, thus eliminating the need to keep track of alias information. Instead, I use alias types extended with index expressions to maintain alias information in the type system.

- I introduce the notion of aggregate types which is used to give types to blocks of memory with heterogeneous types.

With these two modifications it becomes possible to give a precise type to a procedure that takes an array and initialises it with zeros.

In the following chapters I refer to the type systems in Section 3.5 and Section 3.6 as the *extended type system* as the two extensions are orthogonal.

Chapter 4

Examples

In this chapter I work through a number of Featherweight DSP examples with type annotations from the type systems presented in Chapter 3. I describe how the type rules are used to check these annotations. I discuss some of the limitations of the type systems revealed by the examples. Finally, I describe the kind of code that cannot be handled by the type systems.

4.1 Worked Examples

This section presents five Featherweight DSP examples annotated with types from Chapter 3. Each example ends with a section summarising the points that the example illustrates. Three of the examples use the baseline type system from Section 3.2, one example uses the extended type system from Section 3.6, and one example uses a combination of the extended type systems from Section 3.5 and Section 3.6. All examples use ASCII notation for the type annotations. Guided by the statistics from Chapter 2 all examples contain at least one `do`-loop and none of the examples contain any other control transfer instructions, such as `jmp`, except `ret`.

4.1.1 Pointwise Vector Multiplication

Figure 4.1 shows the pointwise vector multiplication code from Figure 2.7 (on page 25) this time with type annotations. The type annotations are drawn from the baseline type system from Section 3.2.

The code for `vecpmult` in Figure 4.1 contains two state type annotations: the first at the entry-point to the procedure, starting at line 2 and ending at line 17; and the second at the top of the body of the `do`-loop, starting at line 19 and ending at line 33.

The state type at the entry-point specifies that to call `vecpmult` the machine must be in a state where `i0` is a pointer to an array of fixed point numbers in X memory of size `n`; `i4` is a pointer to an array of fixed point numbers in Y mem-

```
1   vecpmult:
2       (s, r)
3       { n : int  |  n > 0 }
4       [ i0  : fix xarray(n),
5         i4  : fix yarray(n),
6         i1  : fix xarray(n),
7         i7  : int(n),
8         dsp : r,
9         csp : [ x0  : junk, y0 : junk, a0 : junk,
10                i0  : fix xarray(0),
11                i4  : fix yarray(0),
12                i1  : fix xarray(0),
13                i7  : int(n),
14                dsp : r,
15                csp : s
16              ] :: s
17      ]
18      do (i7) {
19         { n : int, k : int  |  n > 0 /\ 0 <= k < n}
20         [ i0  : fix xarray(n-k),
21           i4  : fix yarray(n-k),
22           i1  : fix xarray(n-k),
23           i7  : int(n),
24           dsp : int(k) :: r,
25           csp : [ x0  : junk, y0 : junk, a0 : junk,
26                   i0  : fix xarray(0),
27                   i4  : fix yarray(0),
28                   i1  : fix xarray(0),
29                   i7  : int(n),
30                   dsp : r,
31                   csp : s
32                 ] :: s
33         ]
34         x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
35         a0=x0*y0
36         xmem[i1] = a0; i1+=1
37      }
38      ret
```

Figure 4.1: Pointwise vector multiplication with type annotations.

ory of size n; i1 is a pointer to an array of fixed point numbers in X memory also of size n; i7 must contain the integer n; the contents of the do-stack, dsp, is denoted by the stack variable r; and finally, the call-stack, csp, contains at least one address (the return address). The type of the return address specifies that x0, y0, and a0 may contain arbitrary values, that i0, i4, and i1 will contain pointers to arrays of size zero, that i7 and dsp will be unchanged, and finally that the return address has been popped from the call-stack upon return.

Even though the type-annotations in Figure 4.1 take up a lot of space they are not complete. I have left out the specification of unchanged registers. To each unchanged register we must assign a new type variable which must be the same in both the state type for the entry point and the state type for the return address. The type of dsp is treated this way in Figure 4.1.

In the following explanation I leave out the store type Ψ because the type correctness of vecpmult is independent of the rest of the store. Futhermore, I use:

- ϕ_1 to denote the index context $\{n : \texttt{int} \mid n > 0\}$;
- ϕ_2 to denote the index context $\phi_1 \wedge \{k : \texttt{int} \mid 0 \leq k < n\}$;
- Δ to denote the type variable context s, r;
- R_1 to denote the register file part of the state type at the entry-point of vecpmult (starting at line 4 ending on line 17);
- R_2 to denote the register file part of the state type at the top of the body of the loop (starting at line 20 ending on line 33);
- and R_4 to denote the regfile type which is the top of the stack type for csp in R_1, that is, the return address (starting at line 9 ending on line 16).

To check the type for the do-loop according to rule (**do**) from Figure 3.9 on page 47 we split the verification into two parts:

- *Entry of the loop:* We must check that the type of the entry point for vecpmult is compatible with the type for the body of the do-loop. That is, first we must check that $R_1 \vdash \texttt{i7} : \texttt{int}(e)$ and $\phi_1 \models e > 0$. Both conditions are trivial because e is n. Second, we must check that a substitution θ_1 exists such that $\phi_1 \vdash \theta_2 : \phi_2$ and $\Delta; \phi_1 \models R_1\{\texttt{dsp} : \texttt{int}(0) :: r\} <: R_2[\theta_1]$. The substitution $[n \mapsto n, k \mapsto 0]$ satisfies the requirements for θ_1.

  Note that the (**do**) rule needs no substitution for type variables as opposed to, for example, the rule (**jmp**) for jumps.

- *The body of the loop:* Before the body of the loop the contents of the registers are specified by the regfile type R_2. After the body of the loop the contents of the registers are described by the regfile type R_3:

```
[ x0  : fix, y0 : fix, a0 : fix,
  i0  : fix xarray(n-k-1),
```

$$\dfrac{\dfrac{\dfrac{\Delta, \phi_3' \models \texttt{fix} \equiv \texttt{fix} \qquad \phi_3' \models n-k-1 = n-k_2}{\Delta, \phi_3' \models \texttt{fix xarray}(n-k-1) <: \texttt{fix xarray}(n-k_2)}}{\Delta, \phi_3' \models R_3(\texttt{i0}) <: R_2[\theta_2](\texttt{i0})}}{\Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]}$$

Figure 4.2: Part of the derivation for $\Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]$, just for the register `i0`

```
i4  : fix yarray(n-k-1),
i1  : fix xarray(n-k-1),
i7  : int(n),
dsp : int(k) :: r,
csp : R4 :: s ]
```

The index context after the body of the loop is the same as at the top of the body, that is, ϕ_2. Let ϕ_3' be the index context $\phi_2 \wedge \{k_2 : \texttt{int} \mid k < n-1 \wedge k_2 = k+1\}$. Now we must check that a substitution θ_2 exists such that $\phi_3' \vdash \theta_2 : \phi_2$ and $\Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]$. The substitution $[n \mapsto n, k \mapsto k_2]$ satisfies the requirements for θ_2.

Part of the derivation for $\Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]$ is shown in Figure 4.2.

Finally, we must check that the state type after the loop is compatible with the type of the return address. This boils down to checking that

$$\phi_2 \wedge k = n \models 0 = n - k$$

which is true.

Points illustrated

While this simple example shows the strengths of the baseline type system, the example also underscores the weaknesses pointed out in Section 3.4. For instance, as the type system does not track alias-information, we cannot tell if the two pointers in `i0` and `i1` have been swapped.

4.1.2 Initialisation of Array

Figure 4.3 shows the procedure `fill_zero` from Figure 3.14 annotated with types. The types used in this example are drawn from the extended type system from Section 3.6, so that alias information is tracked and we can see how the store type changes.

Similar to the code for `vecpmult` the code for `fill_zero` in Figure 4.3 contains two state type annotations: one at the entry-point to the procedure and

```
1   fill_zero:
2       (p, r, s)
3       {n : int | n > 0}
4       XMEM[ p -> junk[n] ]
5       [ i0  : xptr(p, 0)
6       , i11 : int(n)
7       , dsp : r
8       , csp : XMEM[ p -> int(0)[n] ]
9               [ x0  : junk
10              , i0  : xptr(p, n)
11              , i11 : int(n)
12              , dsp : r
13              , csp : s
14              ] :: s
15      ]
16      x0 = 0
17      do (i11) {
18         {..., k : int | ... /\ 0 <= k < n}
19         XMEM[ p -> int(0)[k] @ junk[n-k] ]
20         [ ...
21         , i0  : xptr(p, k)
22         , dsp : int(k) :: r
23         ]
24         xmem[i0] = x0; i0 += 1
25      }
26      ret
```

Figure 4.3: Initialisation of array with type annotations.

one at the top of the body of the `do`-loop. To preserve space and increase readability I have elided all types that are repetitions of previous given types (in this example), marking elided types by ellipses. The code with non-elided types is listed in Appendix A.1.

The state type at the entry-point specifies that to call `fill_zero` the machine must be in a state where there is at least one address `p` in X memory and at `p` there is room for `n` words of memory, but we do not care about the exact location of `p`; `i0` must contain the address `p`; `i11` must contain the integer `n`; and when `fill_zero` returns the memory pointed to by `p` is filled with zeros, `x0` has an undefined value, `i0` is incremented by `n`, and `i11` and `dsp` are unchanged.

In the following I use ϕ_1 to denote the index context $\{n : \texttt{int} \mid n > 0\}$; ϕ_2 to denote the index context $\phi_1 \wedge \{k : \texttt{int} \mid 0 \leq k < n\}$; Δ to denote the type variable context s, r, p; R_1 and Ψ_1 to denote the regfile type and store type parts

of the state type at the entry-point of `fill_zero`; R_2 and Ψ_2 to denote the regfile type and store type parts of the state type at the top of the body of the loop; and R_4 and Ψ_4 to denote the regfile type and store type which is the top of the stack type for `csp` in R_1.

Again, to check the type for the `do`-loop according the rule (**do**) from Figure 3.9 on page 47 (with the suitable modifications adapting it to the extended type system) we split the verification into two parts:

- *Entry of the loop:* We must check that the type of the entry point for `fill_zero`, updated with the extra information that `x0` contains the integer 0, is compatible with the type for the body of the `do`-loop. That is, first we must check that $R_1 \vdash \texttt{i0} : \texttt{int}(e)$ and $\phi_1 \models e > 0$, both are trivial because e is n. Second, we must check that a substitution θ_1 exists such that $\phi_1 \vdash \theta_2 : \phi_2$ and $\Delta; \phi_1 \models R_1\{\texttt{dsp} : \texttt{int}(0) :: r\} <: R_2[\theta_1]$ and $\Delta; \phi_1 \models \Psi_1 <: \Psi_2[\theta_1]$. The substitution $[n \mapsto n, k \mapsto 0]$ satisfies the requirements for θ_1. The tricky part is to check that the aggregate type for `p` in Φ_1 is a subtype of the aggregate type for `p` in $\Psi_2[\theta_1]$. That is, we must check that:

  $$\Delta; \phi_1 \models \texttt{junk}[n] <: \texttt{int}(0)[0]\texttt{@junk}[n-0]$$

  This can be verified using the rules (**aggr-zerol-sub**) and (**aggr-seg-sub**) from Figure 3.20 on page 63.

- *The body of the loop:* Before the body of the loop the contents of the store and the registers are specified by the store type Ψ_2 and the regfile type R_2. After the body of the loop the contents of the store and the registers are described by the store type Ψ_3 and the regfile type R_3:

  ```
  XMEM [ p -> int(0)[k]
           @ junk[k-k] @ int(0)[1] @ junk[n-k-1] ]
  [ x0  : int(0),
    i0  : xptr(p, k+1),
    i11 : int(n),
    dsp : int(k) :: r,
    csp : (Ψ4, R4) :: s ]
  ```

  (these are not listed in Figure 4.3). The index context after the body of the loop is the same as at the top of the body, which is ϕ_2. Let ϕ_3' be the index context $\phi_2 \wedge \{k_2 : \texttt{int} \mid k < n-1 \wedge k_2 = k+1\}$. Now we must check that a substitution θ_2 exists such that $\phi_3' \vdash \theta_2 : \phi_2$ and $\Delta; \phi_3' \models R_3\{\texttt{dsp} : \texttt{int}(k_2) :: r\} <: R_2[\theta_2]$ and $\Delta; \phi_3' \models \Psi_3 <: \Psi_2[\theta_3]$. The substitution $[n \mapsto n, k \mapsto k_2]$ satisfies the requirements for θ_2. Again, the tricky part is to check that the aggregate type for `p` in Ψ_3 is a subtype of the aggregate

type for p in $\Psi_2[\theta_2]$. That is, we must check that:

$$\begin{aligned}\Delta; \phi_3' \models & \texttt{int}(0)[k]\texttt{@junk}[k-k]\texttt{@int}(0)[1]\texttt{@junk}[n-k-1] \\ & <: \\ & \texttt{int}(0)[k_2]\texttt{@junk}[n-k_2]\end{aligned}$$

This can be verified using the rules (**aggr-zerol-sub**), (**aggr-split-sub**) and (**aggr-seg-sub**) from Figure 3.20.

Points illustrated

This example illustrates how the extended type system supports the two key features we sought. First, it is possible to keep track of alias information. With the type annotation we can specify that when `fill_zero` return then `i0` still points to the same array as when `fill_zero` was called, but at the other end of the array. Second, updates of memory locations may change their type. When `fill_zero` returns, the types specify that the array at `p` is filled with zeros. The example also shows how alias types combined with index types can be used to handle pointers into the interior of a memory block in a flexible manner.

4.1.3 Pointwise Vector Multiplication with Prefetch

Figure 4.4 show the procedure `vecpmult_prefetch` from Figure 3.15 with type annotations. The types used in this examples are drawn from the extended type system from Section 3.6. To check the example we also need the rules from Section 3.5 that handle out-of-bound memory reads.

Like the two previous examples, the code for `vecpmult_prefetch` in Figure 4.4 contains two type annotations, one at the entry-point to the procedure and one at the top of the body of the `do`-loop. Again, points of ellipsis mark elided types—the code with non-elided types is listed in Appendix A.2.

The state type at the entry-point of `vecpmult_prefetch` specifies more or less the same as the state type for `vecpmult` in Section 4.1.1. But the pointer and aggregate types also enables us to specify, for instance, that the array pointed to by `i1` contains nonsense values, so that values stemming from `i1` will not be used in computations by mistake.

In the following I use ϕ_1 to denote the index context $\{n : \texttt{int} \mid n > 1\}$; ϕ_2 to denote the index context $\phi_1 \wedge \{k : \texttt{int} \mid 0 \leq k < n\}$; Δ to denote the type variable context s, r, p_1, p_2, p_3; R_1 and Ψ_1 to denote the register file and store type parts of the state type at the entry-point of `vecpmult_prefetch`; R_2 and Ψ_2 to denote the register file and store type parts of the state type at the top of the body of the loop; and R_5 and Ψ_5 to denote the regfile type and store type which is the top of the stack type for `csp` in R_1.

To check the type for the `do`-loop using the rule (**do-oob**) from Figure 3.16 on page 58 (with the suitable modifications adapting it to the extended type system) we split the checking into three parts:

```
1   vecpmult_prefetch:
2       (s, r, p1, p2, p3)
3       {n : int  |  n > 1}
4       XMEM[ p1 -> fix[n], p3 -> junk[n] ]
5       YMEM[ p2 -> fix[n] ]
6       [ i0  : xptr(p1, 0),
7         i4  : yptr(p2, 0),
8         i1  : xptr(p3, 0),
9         i7  : int(n-1),
10        dsp : r,
11        csp : XMEM[ p1 -> fix[n], p3 -> fix[n] ]
12              YMEM[ p2 -> fix[n] ]
13              [ x0  : junk, y0 : junk, a0 : junk,
14                i0  : xptr(p1, n+1),
15                i4  : yptr(p2, n+1),
16                i1  : xptr(p3, n),
17                i7  : int(n),
18                dsp : r,
19                csp : s
20              ] :: s
21      ]
22      x0  = xmem[i0]; i0+=1
23      y0  = ymem[i4]; i4+=1
24      do (i7) {
25         {..., k : int | ... /\ 0 <= k < n}
26         XMEM[ ..., p3 -> fix[k] @ junk[n-k] ]
27         [ ...,
28           x0  : fix,
29           y0  : fix,
30           i0  : xptr(p1, k+1),
31           i4  : yptr(p2, k+1),
32           i1  : xptr(p3, k),
33           dsp : int(k) :: r
34         ]
35         a0=x0*y0; x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
36         xmem[i1] = a0; i1+=1
37      }
38      ret
```

Figure 4.4: Pointwise vector multiplication with prefetch with type annotations.

- *Entry of the loop:* We must check that the type is correct when we enter the loop. This is much like what we had to check for `fill_zero` in the previous example.

- *The body of the loop:* First we must check all but the last round of the loop. This means that when we check the body of the loop we know that `k` is between 0 and $n-1$. Thus, we know that we do not read out of bounds when we read from `i0` and `i4`. Hence, after the body of the loop the contents of the store and the registers are described by the store type Ψ_3 and the regfile type R_3:

  ```
  XMEM[ p1 -> fix[n],
        p3 -> fix[k] @ junk[k-k] @ fix[1] @ junk[n-k-1] ]
  YMEM[ p2 -> fix[n] ]
  [ x0  : fix, y0  : fix, a0  : fix,
    i0  : xptr(p1, k+1+1),
    i4  : yptr(p2, k+1+1),
    i1  : xptr(p3, k+1),
    i7  : int(n),
    dsp : int(k) :: r,
    csp : (Ψ5, R5) :: s
  ]
  ```

  Again, checking that Ψ_3 and R_3 are compatible with Ψ_2 and R_2 is similar to what we did for the loop body in `fill_zero` in the previous example.

- *The last round of the loop:* Finally, we must check the body of loop once more, but now we know that it is the final round. Thus, the index context is now $\phi_2 \wedge k = n-1$, because the index context after the body of the loop is the same as before the body of the loop.

  Now, after the body of the loop the contents of the store and the registers are described by the store type Ψ_4 and the regfile type R_4:

  ```
  XMEM[ p1 -> fix[n],
        p3 -> fix[k] @ junk[k-k] @ fix[1] @ junk[n-k-1] ]
  YMEM[ p2 -> fix[n] ]
  [ x0  : junk, y0  : junk, a0  : fix,
    i0  : xptr(p1, k+1+1),
    i4  : yptr(p2, k+1+1),
    i1  : xptr(p3, k+1),
    i7  : int(n),
    dsp : int(k) :: r,
    csp : (Ψ5, R5) :: s
  ]
  ```

Note that x0 and y0 now contain nonsense values because they have been read outside the arrays. Since we know that $k = n - 1$, Ψ_4 and R_4 can be rewritten to:

```
XMEM[ p1 -> fix[n],
      p3 -> fix[n-1] @ fix[1] @ junk[n-(n-1)-1] ]
YMEM[ p2 -> fix[n] ]
[ x0  : junk, y0  : junk, a0  : fix,
  i0  : xptr(p1, (n-1)+1+1),
  i4  : yptr(p2, (n-1)+1+1),
  i1  : xptr(p3, (n-1)+1),
  i7  : int(n),
  dsp : int(k-1) :: r,
  csp : (Ψ5, R5) :: s
]
```

This is a subtype of (Ψ_5, R_5) when we pop the do-stack and the call-stack.

Points illustrated

Like in the previous example we have seen that alias types and aggregate types really shine when memory reuse is managed explicitly. We are able to specify that vecpmult_prefetch is called with a pointer in i1 to an uninitialised chunk of memory at p3, and when vecpmult_prefetch returns this chunk of memory has been initialised. We have also seen that the out-of-bound rules can be combined with aggregate types, that we are able to read values from memory outside the specified arrays, and that the type system ensures that these values are not used in a computation.

Thus the example illustrates that the two extensions to the baseline type system presented in Section 3.5 and Section 3.6 are orthogonal. The extentions can be combined without interfering with each other and to good effect.

4.1.4 Matrix multiplication

As a larger example with nested do-loops, Figure 4.5 shows a straightforward implementation of matrix multiplication in Featherweight DSP. The code multiplies two matrices, where the first matrix has *rows* rows and col_1 columns and the second matrix has col_2 rows and col_1 columns, thus the result of the multiplication is a matrix with *rows* rows and col_2 columns. The dimensions *rows*, col_1, and col_2 are not known statically and are given to the procedure as arguments. The procedure also takes as argument a pointer to where the result must stored (all arguments are passed in registers).

In this example I use the baseline type system. Again, points of ellipsis mark elided types, and the code with non-elided types is listed in Appendix A.3.1.

```
matrix_mult:
   (s, r)
   {rows : int, col1 : int, col2 : int
         | rows > 0 /\ col1 > 0 /\ col2 > 0}
   [ i6  : int(rows), i7  : int(col1), i8  : int(col2),
     i0  : fix xarray(col1) xarray(rows),
     i1  : fix yarray(col2) yarray(col1),
     i2  : fix xarray(col2) xarray(rows),
     dsp : r,
     csp : [ i6  : int(rows), i7 : int(col1), i8 : int(col2),
             i0  : fix xarray(col1) xarray(0),
             i1  : fix yarray(col2) yarray(col1),
             i2  : fix xarray(col2) xarray(0),
             a0  : junk, x0 : junk, y0 : junk,
             i3  : junk, i4 : junk, i5 : junk,
             i10 : junk, i11 : junk, i12 : junk,
             dsp : r,
             csp : s] :: s
   ]
   do (i6) {
     {..., k1 : int | ... /\ 0 <= k1 < rows }
     [ ...,
       i0  : fix xarray(col1) xarray(rows-k1),
       i2  : fix xarray(col2) xarray(rows-k1),
       dsp : int(k1) :: r
     ]
     i3  = xmem[i0]; i0 += 1
     i5  = xmem[i2]; i2 += 1
     i10 = 0
     do (i8) {
       {..., k2 : int | ... /\ 0 <= k2 < col2 }
       [ ...,
         i3  : fix xarray(col1),
         i5  : fix xarray(col2-k2),
         i10 : int(k2),
         dsp : int(k2) :: int(k1) :: r
       ]
       a0 = 0
       i11 = i1
       i12 = i3
```

Figure 4.5: Matrix multiplication. Part 1

```
41          do (i7) {
42            {..., k3 : int | ... /\ 0 <= k3 < col1 }
43            [ ...,
44              a0  : fix,
45              i11 : fix yarray(col2) yarray(col1-k3),
46              i12 : fix xarray(col1-k3),
47              dsp : int(k3) :: int(k2) :: int(k1) :: r
48            ]
49            i4 = ymem[i11]; i11 += 1
50            x0 = xmem[i12]; i12 += 1
51            i4 = i4 + i10
52            y0 = ymem[i4]
53            a0 += x0 * y0
54          }
55          xmem[i5] = a0; i5 += 1
56          i10 += 1
57        }
58      }
59      ret
```

Figure 4.5: Matrix multiplication. Part 2

The code assumes that registers `i0` and `i1` contain pointers to the two matrices to be multiplied; that register `i2` contains a pointer to where the result should be stored; and that registers `i6`, `i7`, and `i8` contain the dimensions of the matrices. When the code returns, register `i0` and register `i2` contain pointers to arrays of size zero, and the registers `a0`, `x0`, `y0`, `i3`, `i4`, `i5`, `i10`, `i11`, and `i12` contain undefined values.

The code consists of three nested `do`-loops. The outermost loop iterates through the rows of the first argument and the result. The second loop iterates through the columns of the second argument and the result. Notice that we need the loop count for the second loop in a computation (to index into the rows of the second argument). Hence, it is necessary to imitate the loop count in a separate register `i10` (there are no instructions for obtaining the loop count)[1]. The third loop iterates through the rows of the second argument, and computes each element in the result.

Despite the nested loops, the example does not pose any new challenges for the type system. The only novelty is that we see how the stack type for the do-stack, `dsp`, keeps track of the different loop counters. There is no reason to go into more detail.

[1] In the real custom DSP it is possible to obtain the loop count through peripheral addressing. But it is more efficient to just use an imitation register.

The example reveals a shortcoming of the extended type system, namely that it is impossible to express the with the extended type system the type of an array where each element is an array and all the elements are different. This limitation stems from existential quantification, because existential quantification is only allowed over index variables and not location variables. Section 4.2.1 discusses this problem in more detail. The example is also somewhat unrealistic because the matrices are represented as arrays of arrays. Again, this is addressed in Section 4.2.1 where I discuss different representations of matrices and the challenges posed by these representations for the type system.

Points illustrated

This example shows how a more complex example with nested loops can be handled by the baseline type system. Here the index types really shine, because they enforce at compile time that `matrix_mult` is only called with matrices with the right dimensions, and that enough memory has been allocated for the result. Usually in matrix libraries written in high-level languages, the check of dimensions is either performed at runtime or dimensions are not checked at all which can result in violation of memory safety. What is perhaps a bit of surprise—at least it was to me—is that the example also uncovers a defect of the extended type system: that it is impossible to express the type of an array of unshared arrays.

4.1.5 Swapping the contents of two registers

Based on the previous examples in this chapter we might think that it is possible to simplify the (**do**) rule. The substitutions we have needed to check the body of the `do` loops have been almost trivial and quite similar in the examples. The substitutions map every index variable other than k to itself and k, the loop count, to the fresh index variable k_2. An example that requires a more interesting substitution is the contrived procedure `multi_swap`. It swaps the contents of two registers a number of times using a `do`-loop. The code for the `multi_swap` procedure is in Figure 4.6.

The procedure `multi_swap` assumes that register `i0` contains a number n, the number of times the two registers should be swapped, and that `x0` and `y0` are the two registers to swap. Hence, if n is odd the end result is that their contents are swapped; if n is even the contents of the registers are not swapped.

Thus, the substitution we must find to check the body of the loop is:

$$[n \mapsto n, \; p \mapsto q, q \mapsto p \;, x \mapsto x, y \mapsto y, k \mapsto k_2]$$

The interesting part is that p and q are not mapped to themselves.

In the state type in the `do`-loop I have used the modulo operator `mod` in the index context. This is still Presburger arithmetic because modulo and division with a constant can be translated to ordinary Presburger arithmetic. With this

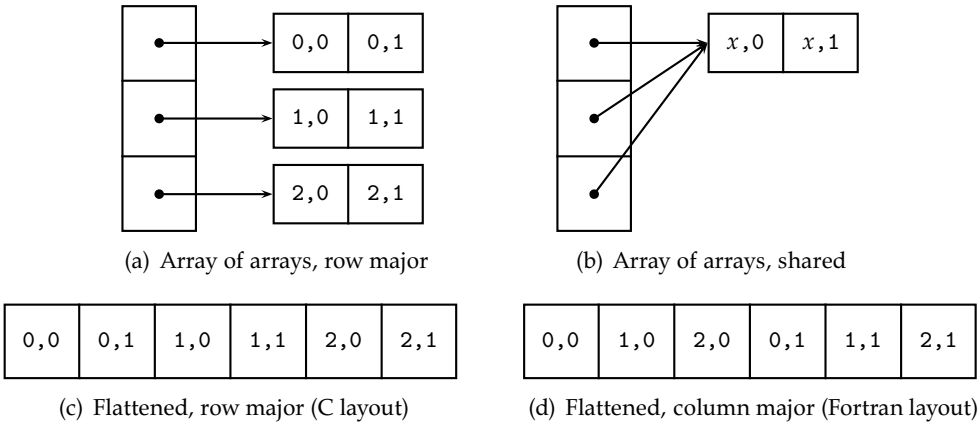
```
multi_swap:
    (r, s)
    {m :int, n : int, p : int, q : int, x : int, y : int
       | n > 0
      /\ (n = 2*m     ==> (x=p /\ y=q))
      /\ (n = 2*m + 1 ==> (x=q /\ y=p)) }
    [ i0  : int(n),
      x0  : int(p),
      y0  : int(q),
      dsp : r,
      csp : [ i0  : int(n),
              x0  : int(x),
              y0  : int(y),
              dsp : r,
              csp : s] :: s
    ]
    do (i0) {
       {... k : int | ... /\ 0 <= k < n
                    /\ (n mod 2  = k mod 2 ==> (x=p /\ y=q))
                    /\ (n mod 2 <> k mod 2 ==> (x=q /\ y=p))}
      }
       [..., dsp : int(k) :: r ]
       x0 = y0; y0 = x0
    }
    ret
```

Figure 4.6: Swapping the contents of two registers in a loop to illustrate the generality of the (**do**) rule.

small extension it is interesting to notice the type annotations form a complete specification of the behaviour of the procedure.

Points illustrated

This example shows the generality of the (**do**) rule. The example illustrates why the (**do**) rule needs to be so complex with all the extra index contexts and substitutions. The example also shows the limits of not including sum types in the type system: it is not possible to give a more general type to `multi_swap` where the contents of `x0` and `y0` are unspecified types and not just integers. This is discussed in more details in Section 4.2.3.



(a) Array of arrays, row major

(b) Array of arrays, shared

(c) Flattened, row major (C layout)

(d) Flattened, column major (Fortran layout)

Figure 4.7: Different representations of matrices

4.2 Limitations of the type system

This section discusses in some detail some of limitations of the type systems from Chapter 3. Some of the limitations where pointed out in Section 4.1.

4.2.1 Representation of Matrices

As already mentioned in Section 4.1.4, matrices pose a challenge for the type systems. Before we go into details about the problems with the type systems, we quickly review the basic strategies for representing matrices (or in general, multi-dimensional arrays) using single-dimensional arrays. One strategy is to represent matrices as arrays of arrays. That is, we represent a matrix with N rows and M columns as an array of size N where each element is a pointer to a row: an array of size M as illustrated in Figure 4.7(a). Alternatively, we can use an array of size M where each element is a pointer to a column: an array of size N. Another strategy is to flatten the matrix into just one array. That is, we represent a matrix with N rows and M columns as an array of size $N \cdot M$. Again, when we flatten the matrix we can either put the elements in row major order, as illustrated in Figure 4.7(c), or in column major order as illustrated in Figure 4.7(d).

In the `matrix_mult` example from Section 4.1.4 we saw that in the baseline type system the regfile type:

```
[ i0 : fix xarray(n) xarray(m) ]
```

specifies that the register `i0` contains a pointer to a matrix with n rows and m columns represented as an array of fixed-point arrays.

When we try to translate this type using the extended type system we run into problems. Our first attempt might be the following store type and regfile type:

```
XMEM[ p1 -> fix [n], p2 -> xptr(p1, 0) [m] ]
[ i0 : xptr(p2, 0) ]
```

However this is not the type of a matrix where each element occupies one distinct memory location. Instead it is the type of a matrix where all the rows are shared, as illustrated in Figure 4.7(b). Notice, that with the baseline type system it is impossible to distinguish between the two representations in Figure 4.7(a) and Figure 4.7(b). One way to fix the extended type system is to introduce existential types over location variables in the style of Walker [53]. If we had existential types over location variables, then we could write the type of a matrix with n rows and m columns represented as an array of fixed-point arrays as the following store type and regfile type:

```
XMEM[ p1 -> (? p . XMEM[ p -> fix [n]] xptr(p, 0)) [m] ]
[ i0 : xptr(p1, 0) ]
```

(where the question mark is used as ASCII notation for existential quantification). Here the regfile type says that `i0` contains the pointer `p1` into X memory, and the store type says that starting at location `p1` there is an block of m pointers, all distinct and different from `p1`, and each pointing to a block of n fixed-point numbers. However, it is not straightforward to introduce existential quantification over location variables, and if we are not careful it is easy to introduce type unsoundness. I go into more detail about how to extend the type system with existential quantification over location variables in Section 6.1.2.

None of our type systems can handle matrices in the flattened representation because of the limitations of Presburger arithmetic. In the baseline type system, if the register `i0` contains a pointer to a matrix with n rows and m columns represented as a single array of fixed-point numbers we would write the regfile type:

```
[i0 : fix xarray(n*m) ]
```

This looks promising, but the index expression denoting the length of the array is not a representable expression in Presburger arithmetics, because we multiply two variables. Similarly, in the extended type system we would write the store type and regfile type as:

```
XMEM[ p1 -> fix [n*m] ]
[ i0 : xptr(p1, 0) ]
```

which suffers from the same problem.

The problem only occurs when both the number of rows and the number of columns are not statically known. For example, if we know that there are three rows and m columns. Then, in the baseline type system, we can write the regfile type:

```
[ i0 : fix xarray(3*m) ]
```

which is expressible in Presburger arithmetic. In the extended type system we have more choices. We could write the store type and regfile type as:

```
XMEM[ p1 -> fix [m] @ fix[m] @ fix[m]]
[ i0 : fix xptr(p1, 0) ]
```

or

```
XMEM[ p1 -> fix [3*m]]
[ i0 : fix xptr(p1, 0) ]
```

which are equivalent, and both types only contain index expressions which are representable in Presburger arithmetic.

We might argue that in resource constrained embedded software without dynamic memory allocation, all dimensions are always statically known, hence the problems described in this section are of no concern. This is a weak argument, however, because we want to be able to type check a procedure without knowing all the places where it is called. And we want library functions like `matrix_mult` to work with different, perhaps dynamically decided, dimensions. Section 6.1.2 discusses other extensions to the type systems.

4.2.2 Size of Type Annotations

It is apparent from the examples shown in this chapter that the type annotations can get large and somewhat unmanageable to write by hand. The two main reasons for the size of the type annotations are:

- *Lack of abstraction.* The type annotations form a partial specification of the meaning of assembler code. Hence, due to the explicit and low level nature of assembler code, the type annotations also need to be quite explicit.
- *Repetition.* As we have seen, large parts of the type annotations are repetitions of types already given. One remedy for this problem may be to introduce syntax for type names to allow type abbreviations in some form.

4.2.3 Type rules not general enough

As noted in Section 3.5 and in Section 4.1.5 the lack of sum types is sometimes cumbersome when we want to make the types less precise. The trouble typically occurs when we want to unify types stemming from different control flows. In Section 3.5 we had to make a more complicated and specialised typing rule for `do`-loops and in Section 4.1.5 we had to settle for less general type annotations.

If we introduce something like the `choose`-types of DTAL, suggested by Xi and Harper [56], we can write a more general type for the entry-point of the `multi_swap` procedure from Section 4.1.5. Figure 4.8 shows the more general polymorphic type for `multi_swap`. But `choose`-types are not enough if we want

```
1   multi_swap:
2       (r, s, a, b)
3       {n : int, m : int, t : int
4          | n > 0
5         /\ (n = 2*m     ==> t = 0)
6         /\ (n = 2*m + 1 ==> t = 1) }
7       [ i0  : int(n),
8         x0  : a,
9         y0  : b,
10        dsp : r,
11        csp : [ i0  : int(n),
12                x0  : choose(t, a, b),
13                y0  : choose(t, b, a),
14                dsp : r,
15                csp : s] :: s
16      ]
```

Figure 4.8: Type annotations for `multi_swap` using `choose`-types.

to type-check `multi_swap` with the more general type. The (**do**) rule only accommodates a substitution over index variable, not type variables. This can be fixed, but the fix requires the type variable context, Δ, to be threaded through all typing rules.

4.3 Comparison to Real Custom DSP Programs

In this section, I relate the examples shown in the previous section with the statistics from Chapter 2.

I have chosen the examples presented in this chapter so that they are somewhat representative for the code style used in the industrial partner's hearing aids (except for the `multi_swap` example). That is, most of the code for the ROM primitives in the industrial partner's hearing aids are based on single `do`-loops used to traverse one or more arrays; so are the examples in this chapter. Thus, I have illustrated how the type systems presented in Chapter 3 can be used to statically check programs written for signal processing in embedded systems.

But the code in the industrial partner's hearing aids uses some additional features of the custom DSP which are not illustrated by examples in this chapter because the type systems does not handle these features:

- *Special purpose addressing modes.* The custom DSP offers two special purpose addressing modes described in Section 2.1.5. The first addressing mode is *modulus addressing* used for *cyclic buffers*. The second addressing mode is *bit-reversing addressing* used for *fast Fourier transformation*. Both of

addressing modes cause problems because they cannot be described by Presburger arithmetic. Section 6.1.2 discusses possible extensions to the type systems that will allow us to work around these problems.

- *Automatic scaling and shifting.* The custom DSP can automatically scale or shift data when they are copied from registers to memory. The type system could be extended to handle this by extending the typing context to track the mode of the custom DSP and mimic the behaviour of the hardware in the typing rules. The cost of this extension is a much more complex typing context and typing rules.
- *Low level interaction with the hardware.* Part of the user code in the industrial partner's hearing aids interacts with the hardware through peripheral space. It is not clear how this should be handled in a general manner, but specialised typing rules for typical usages of peripheral space or a special escape hook from the type system could be introduced.

4.4 Summary

In this chapter I have done two things. First, I have presented several examples in the same coding style as the code for the industrial partner's hearing aids. Then I have shown how the type systems from Chapter 3 can be used to annotate these programs so that it is possible to statically check these programs for certain classes of errors. Second, I have discussed some of the limitations of the type systems which are uncovered by the examples, and suggested how these limitations might be overcome.

Chapter 5

Implementation

This chapter describes a proof-of-concept implementation of a type checker for the type system described in Chapter 3. The purpose is twofold: to describe what are the difficult and novel parts of the implementation, and to enable experimentation to check that using a Presburger solver is feasible in practise for DSP assembler programs.

5.1 Overview of the Checker

The proof-of-concept implementation described in this chapter is only a type checker. The implementation does not reconstruct any nontrivial type annotations through inference. This means that to type check a program, the program must have type annotations at all labels, at the top of the body of all `do`-loops, and after all `call` instructions. That is, all the places that can be targets for control transfer instructions. Section 5.4.1 describes some further restrictions to simplify the implementation.

The typing rules for the judgements in Chapter 3 are mostly syntax directed, so it is straightforward to derive an ML implementation that checks whether the rules are satisfied. In the following I go through the places where the rules are not directly syntax directed. These are the places were some creativity is required in the type checker.

In the rest of the chapter the generic term *type* is used to mean either a store type, a state type, a regfile type, a stack type, an aggregate type, or a plain type τ (see Figure 3.1 and Figure 3.17), and t is used to range over these.

Because the baseline type system and the extended type systems are similar, it is possible to produce a single implementation that contains all three type systems. This supports the claim that the out-of-bounds typing rules from Section 3.5 really are orthogonal to the pointer arithmetic and aggregate object typing rules from Section 3.6.

5.1.1 Instructions and instruction sequences

The judgements for instructions and instruction sequences in in Chapter 3 are mostly directed by the syntactic structure of the instructions. This means that we can systematically derive an SML implementation from the rules. With one function for each judgement, where the body of such a judgement-function is a big case-expression with one clause for each typing rule, and where the pattern of each clause corresponds to instruction in the conclusion of a typing rule.

The typing rules for instructions and instruction sequences in Figures 3.8, 3.9, and 3.11 are interesting challenges to implement because of three features: the threading of typing context, not completely syntax directed for all instructions, and the need to check composite instructions.

Figure 5.1 shows an extract of the implementation of the type checker in pseudo-ML. Here we can see that the basic skeleton of the checker is still based on the syntactic structure of instructions.

Threading of Typing Context

It is somewhat unusual that a typing context is threaded around like in the rules for instructions and instruction sequences in Figures 3.9 and 3.11. Informally we say that the rules take a typing context and return (part of) a new typing context. Remember that a typing context is a type variable context, an index variable context, and a machine configuration type. This threading of typing context reflects the fact that the type system is *control flow sensitive*. This control flow sensitivity is evident in the part of the implementation for the rule (**seq**) where the typing context is threaded, and in the implementation of the rule (**write**) where we use the typing context to check for out off bounds errors.

Overloaded Instructions

For some of the instructions the syntax alone is not enough to determine which variant of the instruction we are dealing with, hence the type rules are not directly syntax directed for these instructions. Consider the instruction:

$$r_1 \ = \ r_2 \ + \ r_3$$

From its syntax alone, we cannot determine whether we are adding two integers, adding two fixed-point numbers, or adding an integer to a memory address. This is a simple form of *add-hoc polymorphism*, sometimes called *overloading* [6], which cannot be extended by the programmer. To solve the ambiguity we use the typing context to determine which variant of the instruction we are checking. In the example we first check the type of r_2 and r_3, and from these types we determine which type rule should be used.

In Figure 5.1 we can see an example of this kind of overloading resolution in the implementation of the rules (**incr-fix**), (**incr-int**), and (**incr-xarr**). Here all

```
fun small Δ φ Ψ1 R1 sins (Ψ2, R2) =
    case sins of
        rd = xmem[rs]  =>          (* Rule (read-oob) *)
          (case R1(rs) of
                τ1 xarray(e)  =>
                  let val τ2 = if φ ⊨ e > 0 then τ1 else junk
                  in  (Ψ2, R2{rd : τ2})
              | _  => type error)
      | xmem[rd] = rs  =>          (* Rule (write) *)
          (case R1(rd) of
                τ1 xarray(e)  =>
                  let val τ2 = R1(rs)
                  in  if φ ⊨ e > 0 then
                        if Δ;φ ⊨ τ2 <: τ1 then (Ψ2, R2)
                        else subtype error
                      else out off bounds error
              | _  => type error)
      | r += aexp  =>      (* Rules (incr-fix), (incr-int), (incr-xarr) *)
          let val τ1 = typeOf φ Ψ1 R1 aexp
          in  case (R1(r), τ1) of
                  (fix, fix)  => (Ψ2, R2)
                | (int(e1), int(e2)) => (Ψ2, R2{r : int(e1 + e2)})
                | ...
      | ...

fun instruction Δ φ Ψ R ins =
    case ins of
        sins1; ... ; sinsn  =>   (* Rule (comp) *)
          if uniqDef(sins1; ... ; sinsn) then
            let val simp = small Δ φ Ψ R
                val (Ψ',R') = (simp sinsn ∘ ··· ∘ simp sins1) (Ψ,R)
            in  (φ, Ψ', R')
          else race condition error
      | ...

 fun insSeq Δ φ Ψ R I =
    case I of
        ins I'  =>                 (* Rule (seq) *)
          let val (φ', Ψ', R') = instruction Δ φ Ψ R ins
          in  insSeq Δ φ' Ψ' R' I' =
      | ...
```

Figure 5.1: Extract of the implementation of the type checker in pseudo-ML

three rules are implemented in one case-clause and the types of the register r and th arithmetic expression *aexp* are used to resolve the overloading.

Composite Instructions

An interesting detail of the implementation is the simplicity of the part that corresponds to the rule (**par**) in Figure 3.9 for composite instructions and the judgement for small instructions in Figure 3.8. To check a composite instruction:

$$sins_1 \texttt{;} \ldots \texttt{;} sins_n$$

we check each of the small instructions $sins_i$ in the same type context. Each yields a new machine configuration type. All these machine configuration types must then be composed.

This is easily expressed using idiom from functional programing. Here we let the function that checks a small instruction return a part of the composition function. The function `small` that type checks a single small instruction takes a typing context and the small instruction as arguments, and returns a function that takes a machine configuration type as argument and returns a machine configuration type. That is, the function `small` has the signature:

```
val small : Δ -> φ -> Ψ -> R -> sins -> Ψ * R -> Ψ * R
```

(where I am sloppy and use the syntactic categories from Chapter 3 as types). Now the rest of the machine configuration composition function is simply the composition of the intermediate functions returned by `small`.

This trick only works because the wellformedness conditions ensured by UniqDef ensure that no race conditions can occur, that is, each register must be assigned by at most one $sins_i$ in a composite instruction and a single store to each of X and Y memory is allowed.

5.1.2 Subtype check

An interesting part of the implementation is the code for testing the subtype relations, namely to check whether one type t_1 is a subtype of another type t_2 in a given type context, Δ, and index context, ϕ, that is $\Delta; \phi \models t_1 <: t_2$.

The subtype relation rules are all syntax directed except for the parts of the rules that deal with dependent types. Instead of interleaving structural checking of the syntax with checking of Presburger propositions, we perform a subtype check in two phases. First, we translate the check into a Presburger proposition without using the index context ϕ. Second, we check that this proposition is satisfied. That is, $\Delta; \phi \models t_1 <: t_2$ is rewritten to:

$$\phi \models [\![t_1 <: t_2]\!]_\Delta$$

$$
\begin{array}{lll}
[\![\texttt{int}(e_1) <: \texttt{int}(e_2)]\!]_\Delta & \equiv & e_1 = e_2 \\
[\![\tau_1\ \texttt{xarray}(e_1) <: \tau_2\ \texttt{xarray}(e_2)]\!]_\Delta & \equiv & e_1 = e_2 \wedge [\![\tau_1 <: \tau_2]\!]_\Delta \\
[\![\tau_1 <: \exists\phi.\tau_2]\!]_\Delta & \equiv & \exists n_1 \cdots n_m.(P \wedge [\![\tau_1 <: \tau_2]\!]_\Delta) \\
 & & \text{where } \phi \text{ is } \{n_1 : \texttt{int}, \ldots, n_m : \texttt{int} \mid P\} \\
[\![\exists\phi.\tau_1 <: \tau_2]\!]_\Delta & \equiv & \forall n_1 \cdots n_m.(P \Rightarrow [\![\tau_1 <: \tau_2]\!]_\Delta) \\
 & & \text{where } \phi \text{ is } \{n_1 : \texttt{int}, \ldots, n_m : \texttt{int} \mid P\} \\
[\![R_1 <: R_2]\!]_\Delta & \equiv & \bigwedge_{r \in R_2} [\![R_1(r) <: R_2(r)]\!]_\Delta \\
 & & \wedge\ [\![R_1(\texttt{csp}) <: R_2(\texttt{csp})]\!]_\Delta \\
 & & \wedge\ [\![R_1(\texttt{dsp}) <: R_2(\texttt{dsp})]\!]_\Delta
\end{array}
$$

Figure 5.2: Part of the translation of a subtype check into a Presburger formula.

where $[\![t_1 <: t_2]\!]_\Delta$ is the function that translates its arguments to a Presburger proposition based on the syntactic structure of t_1 and t_2. Figure 5.2 shows some of the parts of the definition of this translation, we obmit the many cases where the syntactic structure alone determines the result of the subtype check. For example, $[\![\texttt{int}(e) <: \texttt{fix}]\!]_\Delta$ translates to a false proposition, independently of e. The translation function in Figure 5.2 is in some sense just Figure 3.6 turned sideways, thus I shall not give a complete definition of $[\![_]\!]_\Delta$.

The translation of a subtype check for aggregate types is more involved because the rules for the subtype relation for aggregate types is not syntax directed. Section 5.3 describes the translation for aggregate types.

In the rest of this chapter I leave out the Δ from the $[\![_]\!]_\Delta$ function as it is not interesting. I use $[\![t_1 <: t_2]\!]$ freely to stand for a Presburger proposition.

5.1.3 Substitutions

Another tricky part of the implementation is determining the substitutions that we need when checking the control transfer instructions such as `do` and `jmp`. We might hope to find a most general substitution, so that backtracking can be avoided. Unfortunately, in general, it is impossible to find such most general substitution for index variables. For example, if we need to find an index variable substitution for the index variable n_2 and k_2 such that the two index expressions:

$$n_1 + k_1 \qquad \text{and} \qquad n_2 + k_2$$

are equal, then several substitutions are possible, for example:

$$
\begin{array}{l}
[n_2 \mapsto n_1, k_2 \mapsto k_1] \\
[n_2 \mapsto k_1, k_2 \mapsto n_1] \\
[n_2 \mapsto n_1 + 1, k_2 \mapsto k_1 - 1]
\end{array}
$$

None of these substitutions is more general that the others. Fortunately, examining the type rules carefully, we can observe that the regfile type or a store type

occuring in a conclusion never involves a substitution froma a premise. Hence, we do not need to find the actual substitutions, we only need to check that they exist.

Checking that a type variable substitution Θ exists is straightforward because we are only looking at type checking, not inference. For index variable substitutions, θ, note that whenever we need to find a substitution we have an index context ϕ_1 and machine configuration M_1 and we need to check that these are compatible with another index context ϕ_2 and machine configuration M_2. Thus, there are two constraints that have to be satisfied:

$$\phi_1 \vdash \theta : \phi_2 \qquad \text{and} \qquad \Delta; \phi_1 \models M_1 <: M_2[\theta]$$

These constraints are translated into a single Presburger formula:

$$\forall x_1 \cdots x_n.P_1 \Rightarrow (\exists y_1 \cdots y_m.P_2 \wedge [\![M_1 <: M_2]\!])$$

where $x_1 \cdots x_n$ are the index variables bound by ϕ_1, P_1 is the proposition constraining $x_1 \cdots x_n$, $y_1 \cdots y_m$ are the index variables bound by ϕ_2, and P_2 is the proposition constraining $y_1 \cdots y_m$ (if $\mathbf{dom}(\phi_1)$ and $\mathbf{dom}(\phi_2)$ overlap we first need to do some α-conversion to make them disjunct). We then check this proposition for satisfiability.

5.2 Out of bounds rules

As noted in Section 3.5 the rule (**read-oob**) from Figure 3.16 which allows out-of-bounds memory reads, is special because it introduces type dependency of the index context. That is, the index context not only rules out certain programs, the index context can also determine the syntactic structure of the types in the conclusion of judgements. In other words, with this rule it is no longer possible to erase all index expressions from types and still get a type correct program. This is a departure for DTAL where it is possible to erase all index expressions.

This might sound like we are introducing some nasty complications. But it turns out that this case is similar to the case for overloaded instructions, described in Section 5.1.1, and it can be handled in a straightforward manner as we can see in Figure 5.1. Given an index context ϕ, to check a load instruction:

$$r_d \texttt{ = xmem[}r_s\texttt{]}$$

we proceed as follows: First, we check that r_s has an array type for the right memory bank (X memory in this case), that is, τ_1 $\texttt{xarray}(e)$. Second, we use ϕ to check that we are guaranteed to be inside the bounds of the array, that is, $\phi \models e > 0$. If this check fails, then r_d is given the type `junk`; otherwise r_d is given the type τ_1 in the resulting regfile type. The case for aggregate objects is similar with respect to the out-of-bound issues, but memory reading in the context of aggregate objects is more involved and is described in the following section.

5.3 Pointer Types and Aggregate Types

The two novelties in the extended type system presented in Section 3.6 are pointer types and aggregate types. The checking of pointer types does not introduce any new difficulties, the only new thing we have to handle is that type substitutions must maintain alias information, which I will not describe in detail.

The checking of aggregate types is more challenging. Compared to what we have to handle for the baseline type system there are two new problems that we have to tackle:

- *Index expression dependent types.* In both the (**read-pa**) rule and in the (**write-pa**) rule in Figure 3.21, we need to find the appropriate segment of an aggregate type, and this segment dependents on the index context and an index expression. This situation is similar to the case for out-of-bounds memory reads, which is discussed in the previous section.

- *Subtype checking of aggregate types.* For all the control transfer instructions we need to check that one machine configuration is compatible with another machine configuration. This, in turn, means that we have to check that one store type is a subtype of another store type. This requires a point-wise check that an aggregate type is a subtype of another aggregate type. What makes this tricky is that the subtype relation for aggregate types is rich, since we are allowed to split and join segments based on equality for Presburger expressions.

5.3.1 Segments

In both the (**read-pa**) rule and in the (**write-pa**) rule in Figure 3.21, we need to find the appropriate segment $\tau_i[e_i]$ of an aggregate type $\tau_1[e_1]@\cdots@\tau_n[e_n]$ with respect to an index context ϕ and an index expression e: The segment $\tau_i[e_i]$ that contains element number e of the aggregate. To find the index i we translate the problem into n propositions:

$$
\begin{aligned}
&0 \leq e < e_1 \\
&0 \leq e - e_1 < e_2 \\
&\vdots \\
&0 \leq e - (e_1 + \cdots + e_{n-1}) < e_n
\end{aligned}
$$

and find the one that is satisfied in ϕ. There is at most one index i for which the corresponding proposition is satisfied if all the e_js are strictly larger than zero, but it is not guaranteed that such an index i exists. We can ensure that all the e_js are strictly larger than zero in the generated propositions if we first filter out all the segments that have size zero (recall that no wellformed segment has a size less than zero).

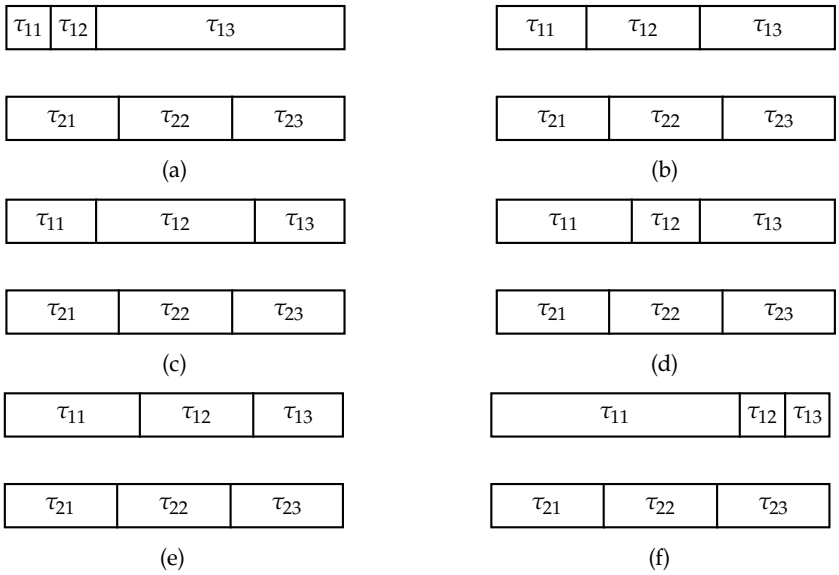


Figure 5.3: The six general cases for matching two aggregate types, each with three segments.

For rule (**write-pa**) in Figure 3.21 we must make sure that we only construct aggregate types where all the segments have a size that is greater than or equal to zero.

5.3.2 Subtype checking of aggregate types

In Section 5.1.2 I stated that the subtype check for aggregate types is not as straightforward as the subtype check for the other kinds of types because the rules for the subtype relation for aggregate types of Section 3.6.3 is not syntax directed.

As preparation for the description of how to translate a subtype check for aggregate types to a Presburger proposition, I present a medium sized example, to illustrate how the translation works.

The example we shall go through is how to build a Presburger formula for the aggregate subtype check of the form:

$$\tau_{11}[x_1]@\tau_{12}[x_2]@\tau_{13}[x_3] <: \tau_{21}[y_1]@\tau_{22}[y_2]@\tau_{23}[y_3]$$

(here we use x_i and y_i rather than e_i to stand for index expressions, so that it is easier in the following to track where the different expressions come from). Figure 5.3 shows the six general cases for how the two aggregate types can match up.

We build the formula from four large subformulae, one subformula for the global structure of the two aggregate types and one for each segment, in this case three, of the aggregate type on the left-hand side. Each subformulae for

the segments consists of a number of smaller subformulae, namely one for each segment of the aggregate type on the right-hand side.

- First of all, the size of the two aggregate types must be the same:

$$x_1 + x_2 + x_3 = y_1 + y_2 + y_3.$$

- The first segment, $\tau_{11}[x_1]$, on the left-hand side: First, we note that in all the cases of Figure 5.3 τ_{11} must be a subtype of τ_{21}, if both x_1 and y_1 are strictly greater than zero. That is:

$$x_1 > 0 \Rightarrow (y_1 > 0 \Rightarrow [\![\tau_{11} <: \tau_{21}]\!]).$$

  Also, if x_1 is larger than y_1 then τ_{11} must be a subtype of τ_{22}. This corresponds the the cases (d), (e), and (f). Again only if both x_1 and y_2 are strictly greater than zero. That is:

$$x_1 > 0 \Rightarrow (y_2 > 0 \Rightarrow (x_1 > y_1 \Rightarrow [\![\tau_{11} <: \tau_{22}]\!])).$$

  Finally, if x_1 is larger than the sum of y_1 and y_2 then τ_{11} must be a subtype of τ_{23}, corresponding to case (f). That is:

$$x_1 > 0 \Rightarrow (y_3 > 0 \Rightarrow (x_1 > y_1 + y_2 \Rightarrow [\![\tau_{11} <: \tau_{23}]\!])).$$

- The second segment, $\tau_{12}[x_2]$, on the left-hand side: First, if the sum of the sizes of the previous segments, in this case only x_1, is strictly smaller than y_1 then τ_{12} must be a subtype of τ_{21}, corresponding to the cases (a), (b), and (c). That is:

$$x_2 > 0 \Rightarrow (y_1 > 0 \Rightarrow (x_1 < y_1 \Rightarrow [\![\tau_{12} <: \tau_{21}]\!])).$$

  Second, if the sum of the sizes of the previous segments is strictly smaller than the sum $y_1 + y_2$ and if x_2 is strictly larger than the difference between the sum of the sizes of the previous segments on the right-hand side, that is y_1, and the sum of the sizes of the previous segments on the left-hand side, still just x_1, then τ_{12} must be a subtype of τ_{22}, corresponding to the cases (b), (c), (d), and (e). That is:

$$x_2 > 0 \Rightarrow (y_2 > 0 \Rightarrow ((x_1 < y_1 + y_2 \wedge x_2 > y_1 - x_1) \Rightarrow [\![\tau_{12} <: \tau_{22}]\!])).$$

  Finally, if the sum of the sizes of the previous segments is strictly smaller than the sum $y_1 + y_2 + y_3$ and if x_2 is strictly larger than the difference between the sum of the sizes of the previous segments on the right-hand side, that is y_1 and y_2, and the sum of the sizes of the previous segments

on the left-hand side, then τ_{12} must be a subtype of τ_{23}, corresponding to the cases (c), (e), and (f). That is:

$$x_2 > 0 \Rightarrow (y_3 > 0 \Rightarrow ((x_1 < y_1 + y_2 + y_3 \wedge \\ x_1 + x_2 > y_1 + y_2) \Rightarrow [\![\tau_{12} <: \tau_{23}]\!])).$$

- The last segment, $\tau_{13}[x_3]$, on the left-hand side: First, if the sum of the sizes of the previous segments, in this case x_1 and x_2, is strictly smaller than y_1 then τ_{13} must be a subtype of τ_{21}, corresponding to the case (a). That is:

  $$x_3 > 0 \Rightarrow (y_1 > 0 \Rightarrow (x_1 + x_2 < y_1 \Rightarrow [\![\tau_{13} <: \tau_{21}]\!])).$$

  Second, if the sum of the sizes of the previous segments is strictly smaller than the sum $y_1 + y_2$ then we know that x_3 is strictly larger than the difference between the sum of the sizes of the previous segments on the right-hand side and the sum of the sizes of the previous segments on the left-hand side (the two aggregate types have the same total size). Thus, τ_{13} must be a subtype of τ_{22}, corresponding to the cases (a), (b), and (d). That is:

  $$x_3 > 0 \Rightarrow (y_2 > 0 \Rightarrow (x_1 + x_2 < y_1 + y_2 \Rightarrow [\![\tau_{13} <: \tau_{22}]\!])).$$

  Finally, if x_3 is strictly larger than zero, then in all the cases the sum of the sizes of the previous segments on the left-hand side is strictly smaller than the sum of all the sizes of the segments on the right-hand side. Thus, τ_{13} must be a subtype of τ_{23}. That is:

  $$x_3 > 0 \Rightarrow (y_3 > 0 \Rightarrow [\![\tau_{13} <: \tau_{23}]\!]).$$

In Figure 5.4 all the subformulae from the example have been assembled. In the example I silently made some simplifications on the fly where I left out redundant parts of subformulae, for the sake of presentation. These simplifications make the large formula in Figure 5.4 appears non-uniform.

However, all the subformulae can be derived in a uniform way. This translation correspond to the second step for the second segment on the left-hand side, that is, the part where we calculated the conditions for when τ_{12} should be a subtype of τ_{22}. Hence, in the general case:

$$\tau_{11}[x_1]@\cdots@\tau_{1n}[x_n] <: \tau_{21}[y_1]@\cdots@\tau_{2m}[y_m]$$

Given a segment on the left-hand side, $\tau_{1i}[x_i]$, we want to describe the conditions for when it matches part of a given segment on the right-hand side, $\tau_{2j}[y_j]$. Described in natural language the conditions are that if the sum of the sizes of the previous segments on the left-hand side is strictly smaller than the sum

$$
\begin{aligned}
& x_1 + x_2 + x_3 = y_1 + y_2 + y_3 \\
\wedge \\
& (x_1 > 0 \Rightarrow (y_1 > 0 \Rightarrow [\![\tau_{11} <: \tau_{21}]\!]) \\
& \qquad \wedge (y_2 > 0 \Rightarrow (x_1 > y_1 \Rightarrow [\![\tau_{11} <: \tau_{22}]\!])) \\
& \qquad \wedge (y_3 > 0 \Rightarrow (x_1 > y_1 + y_2 \Rightarrow [\![\tau_{11} <: \tau_{23}]\!]))) \\
\wedge \\
& (x_2 > 0 \Rightarrow (y_1 > 0 \Rightarrow (x_1 < y_1 \Rightarrow [\![\tau_{12} <: \tau_{21}]\!])) \\
& \qquad \wedge (y_2 > 0 \Rightarrow ((x_1 < y_1 + y_2 \wedge x_1 + x_2 > y_1) \Rightarrow [\![\tau_{12} <: \tau_{22}]\!])) \\
& \qquad \wedge (y_3 > 0 \Rightarrow ((x_1 < y_1 + y_2 + y_3 \wedge x_1 + x_2 > y_1 + y_2) \Rightarrow [\![\tau_{12} <: \tau_{23}]\!]))) \\
\wedge \\
& (x_3 > 0 \Rightarrow (y_1 > 0 \Rightarrow (x_1 + x_2 < y_1 \Rightarrow [\![\tau_{13} <: \tau_{21}]\!])) \\
& \qquad \wedge (y_2 > 0 \Rightarrow (x_1 + x_2 < y_1 + y_2 \Rightarrow [\![\tau_{13} <: \tau_{22}]\!])) \\
& \qquad \wedge (y_3 > 0 \Rightarrow [\![\tau_{13} <: \tau_{23}]\!]))
\end{aligned}
$$

Figure 5.4: The Presburger formula for checking the subtype relation for two aggregate types, each with three segments.

$$
\begin{aligned}
& [\![\tau_{11}[x_1] @ \cdots @ \tau_{1n}[x_n] <: \tau_{21}[y_1] @ \cdots @ \tau_{2m}[y_m]]\!]_\Delta \equiv \\
& \sum_{i=1}^{n} x_i = \sum_{j=1}^{m} y_j \\
\wedge \\
& \bigwedge_{i=1}^{n} \Big(x_i > 0 \Rightarrow \bigwedge_{j=1}^{m} \Big(y_j > 0 \Rightarrow ((\sum_{k=1}^{i-1} x_k < \sum_{l=1}^{j} y_l) \wedge (\sum_{k=1}^{i} x_k > \sum_{l=1}^{j-1} y_l) \Rightarrow [\![\tau_{1i} <: \tau_{2j}]\!]) \Big) \Big)
\end{aligned}
$$

Figure 5.5: Translation of a subtype check of aggregate types to a Presburger formula.

sizes of the previous segments on right-hand side plus y_j, and if the sum of the sizes of the previous segments on the left-hand side plus x_i is strictly larger than the sum sizes of the previous segments on right-hand side, and if x_i and y_j are larger than zero, then τ_{1i} should be a subtype of τ_{2j}. In Figure 5.5 the translation of a subtype check for aggregate type into a Presburger formula is formalised.

5.4 Checking Presburger Formulae

This section describes the techniques used to implement the satisfiability check for Presburger formulae. The solver is based on Norrish's implementation of the Omega-test, which is part of the HOL4 theorem prover. It is outside the scope of this dissertation to give a complete description of the Omega-test and Norrish's implementation.

The Omega-test as described in [44] is an extension of Fourier–Motzkin variable elimination [14]. But while Fourier–Motzkin variable elimination is incomplete for integer problems the Omega-test is complete for integer problems.

Norrish's implementation of the Omega-test consists of two parts: a *core engine* outside the HOL logic, and a library and a theory inside the HOL logic. I am using the core engine which is written as a library in SML and is independent of the rest of HOL. The core engine can find satisfying assignments to formulae on the form:

$$\begin{aligned}\exists x_1 x_2 \ldots x_n.\\ &0 \leq c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}x_n\\ \wedge\ &0 \leq c_{21}x_1 + c_{22}x_2 + \cdots + c_{2n}x_n\\ \wedge\ &\vdots\\ \wedge\ &0 \leq c_{m1}x_1 + c_{m2}x_2 + \cdots + c_{mn}x_n\end{aligned} \qquad \textbf{(core)}$$

where the x_is are variables and the c_{ij}s are integer constants. We say that formulae on the form (**core**) are on *core form*, and use C to denote the syntactic category of formulae of the form:

$$0 \leq c_{i1}x_1 + c_{i2}x_2 + \cdots + c_{in}x_n.$$

Now note that all the formulae that we have to check are of one of two forms:

- $\forall x_1 \cdots x_n.P_1 \Rightarrow P_2$, stemming from the judgement $\phi \models P_2$, where $x_1 \cdots x_n$ are the index variables bound by ϕ and P_1 is the proposition constraining these, described in Section 3.1.5.

  To check a formula on this form, we first rewrite it to:

  $$\neg\exists x_1 \cdots x_n.\neg(P_1 \Rightarrow P_2)$$

  then rewrite $\neg(P_1 \Rightarrow P_2)$ to disjunctive normal form and distribute the existential on each disjunct:

  $$\neg((\exists x_1 \cdots x_n.C_1) \vee \cdots \vee (\exists x_1 \cdots x_n.C_m))$$

  Now each disjunct is of core form and we can use the core engine. If we find a satisfying assignment for one of the disjuncts then we have found

a counterexample for the original formula; otherwise the original formula is true.

For this form I assume that there are no universal or existential quantifiers in P_1 and P_2. If there are quantifiers in P_1 or P_2, we would be in a situation similar to the next case.

- $\forall x_1 \cdots x_n.P_1 \Rightarrow (\exists y_1 \cdots y_m.P_2 \wedge [\![t_1 <: t_2]\!])$, stemming from substitutions as described in Section 5.1.3.

  To check a formula of this form we first eliminate the existential quantifier, and the quantifiers that might be in the formula $[\![t_1 <: t_2]\!]$ and then proceed as in the previous case.

5.4.1 Elimination of Existential Quantifiers

The elimination of the existential quantifier in the second form, however, is not straightforward. Pugh and Wonnacott [45] describes how elimination of existential quantifiers is implemented in thier Omega-test. In Norrish's implementation of the Omega-test the part the does elimination of existential quantifiers is implemented inside the HOL logic. Hence, it is not possible to directly reuse Norrish's implementation in my solver.

I have only implemented a crude and incomplete elimination algorithm: If we want to eliminate the existential quantifier $\exists x.P$ we check whether P is on the form $(x = e) \wedge P'$ (perhaps after using some associative and commutative rewrites) and if it is then we substitute e for x in P', thus eliminating x. While this is simple, it is sufficient to check four of the five examples in Chapter 4. The example that cannot be checked is the `multi_swap` procedure, the checking fails because I have not implemented support for modulo with constants in my checker.

Furthermore, to simplify the implementation I impose the following restrictions on type annotations:

- no existential types are allowed,
- only the top-level state type annotations at labels are allowed to introduce new type variables (but new *index variables* can be introduced almost anywhere),
- and finally nested state types are not allowed to introduce new type variables nor new index variables.

The lack of proper quantifier elimination is the reason why I have imposed these restrictions, because they eliminate nested quantifiers stemming from existential types and state types, thus we only have to deal with the quantifiers stemming from the need to check that substitutions exists when checking control transfer instructions as described in Section 5.1.3.

Program	Code (# lines)	Types (# lines)	Total (# lines)	Parse (sec.)	Check (sec.)
`vecpmult`	7	22	29	0.033	0.013
`fill_zero`	6	21	27	0.030	0.015
`vecpmult_prefetch`	8	32	40	0.049	0.056
`matrix_mult`	21	35	56	0.084	0.136
`matrix_mult_extended`	21	53	74	0.106	0.191
`add_im_part`	7	16	23	0.029	0.009
`add_im_part_extended`	7	18	25	0.032	0.012
`sum_re_im_part`	9	19	28	0.041	0.031
`all_repeated_six`	516	1296	1812	2.326	2.580

Figure 5.6: Benchmark numbers.

5.5 Benchmarks

One of the reasons for making the proof-of-concept implementation described in this chapter is to test the thesis that an implementation based on a solver for Presburger arithmetic is feasible in practise to use for handwritten DSP assembler code. Practical feasibility means two things here: that the programmer does not have to write an inordinate amount of type annotations, and that the type checker does not have to use too much time or too much memory to check a type annotated program. Both of these notions of practical feasibility are deliberately vague, because they are largely dependent on taste and situation.

To test the thesis I have made a small benchmark suite to obtain performance numbers and to form an idea of how many type annotations are needed. Figure 5.6 shows the benchmark numbers for this small suite of example programs. In the figure I report the number of lines of code and the number of lines taken up by type annotations. Strictly speaking, the number of lines taken up by type annotations is not a meaningful measure because type annotations are not line oriented, but I have tried to write the type annotations in a natural style, so that the numbers are somewhat meaningful.

The first four programs in Figure 5.6 are the first four examples from Chapter 4. The other programs are:

- `matrix_mult_extended` is a version of the matrix multiplication example but with type annotations from the extended type system. Thus, each of the three matrices only contains one shared row, according to the type system. The code for this example can be found in Appendix A.3.2.

- `add_im_part` takes an array of complex numbers represented as an flat array of fix point numbers as input, and sums all the imaginary parts, that is, all the elements on the odd indexes. The code uses the prefetch

idiom. The type annotations are drawn from the baseline type system. See Appendix A.4.1.

- `add_im_part_extended` is the same program as `add_im_part`, but uses the extended type system for type annotations. See Appendix A.4.2.
- `sum_re_im_part` is similar to `add_im_part`. The code sums both the real parts and the imaginary parts of an array of complex numbers. The code uses the prefetch idiom and makes two out-of-bounds reads. See Appendix A.5.
- `all_repeated_six` is all the programs above repeated six times, with labels suitably renamed, in one program. This is an attempt to get a rough measurement of how long time it would take to check the entire corpus of ROM primitives in the industrial partner's hearing aids.

  This program has 48 procedures whereas there are 43 ROM primitives. But this program only takes up 512 lines of code (not including type annotations) whereas the ROM primitives take up 1074 lines of code. On the other hand this program has more procedures with nested `do`-loops (all the matrix multiplications procedures) than the corpus of ROM primitives.

For each of the performance benchmarks I ran the test in a loop 2000 times and then found the average running time. Hence the reported times are probably a bit slower than what they would be in a one-shot run because of garbage collection. All tests were performed on my lightly loaded IBM thinkpad, with a 733 MHz Intel Pentium III (Coppermine) processor and 384 Mb RAM, running Linux. None of the examples needed more than 4 Mb to be checked.

The performance numbers in Figure 5.6 strongly suggest that a type checker based on a Presburger solver is practically feasible for handwritten DSP assembler programs. It is stunning that the example programs are checked as fast as they are read from disk and parsed, albeit the parser is implemented with backtracking parser combinators and not optimised in any way. If the `all_repeated_six` example program really is representative of the ROM primitives, then we can check the entire corpus of the industrial partner's ROM primitives in about ten seconds (including the reading and parsing of files).

On the other hand, the amount of type annotations looks a bit daunting. Roughly speaking there are almost three times as many lines of type annotations as there are code lines. This is not to surprising as little has been done to make the type annotations smaller, through syntactic sugar, for instance.

5.6 Summary

In this chapter I have described the interesting parts of my proof-of-concept SML implementation of a type checker for the type system and extensions pre-

sented in Chapter 3. We have seen how to handle the parts of the judgements in Chapter 3 that are not syntax directed. In particular we have seen how to translate a subtype check into a pure Presburger proposition, also for aggregate types, which can then be checked for satisfiability. Finally, I have presented some benchmark numbers that suggest that a type checker based on a Presburger solver is practically feasible for handwritten DSP assembler programs. This implementation allows us to make a quantitative measurement of the type annotations in whereas the measurement in the previous chapter was qualitative.

Chapter 6

Future Work and Related Work

This chapter evaluates the work presented in the previous three chapters, suggests how this work can be extended, and compares the work with related research.

6.1 Future Work

In the previous three chapters I have shown that DTAL's type system can be adapted for Featherweight DSP, and that this type system can be used for giving type annotations for handwritten DSP assembler code using common idioms such as memory prefetch, and that these annotations can be automatically checked. The main goal for this work is to test the thesis presented in Section 1.1, but the work is also interesting on its own. This section describes how the work in the previous chapters could be extended.

6.1.1 Extentions to Featherweight DSP

Featherweight DSP from Section 2.3 does not handle all the features of the real custom DSP. For instance, the modulo and reverse binary addressing modes (see Section 2.1.5) are not handles. The real custom DSP also supports:

- Reading and writing to the same address in both X and Y memory in one instruction. For example, the instruction:

  ```
  x0,y0 = xymem[i4]
  ```

  loads the data at `i4` in X memory into the register `x0` and the data at `i4` in Y memory into register `y0`. This is useful for representing an array of complex numbers where the real part is in X memory and the imaginary part is in Y memory, for instance.

  It would not be too much work to extend the abstract machine, type systems, and implementation with this feature.

- Multiple labels into the same instruction sequence. It is, for example, common for procedures in assembly language to have multiple entry points. It would not be a lot of work to allow this in the syntax of Featherweight DSP and the abstract machine could also be extented to handle this. The important property to preserve is that jumps into the body of a `do`-loop are disallowed, that is, labels in the body of a `do`-loop is not allowed. Similar, the type system and implementation can be extended to handle multiple labels in the same instruction sequence. The problem here is to minimise how many type annotations are needed.

- Auto increment when indirect addressing is used in control transfer instructions. For example:

  ```
  jmp(i0); i0 += 1
  ```

  The abstract machine for Featherweight DSP can easily be extended to handle this feature. But it is not clear how easily this feature could be handled in the type system. This feature is not used in the industrial partner's code, hence I have not been motivated to work out the details.

6.1.2 Extentions to the Type Systems

The DTAL baseline type system in Section 3.2 and the alias types used the extended type system in Section 3.6 have been stripped to the most essential features to simplify the type systems, to ease the implementation, and to get a better understanding of which features of the type systems are the most essential ones. This section examines which features would be nice to add to the type system, and what we would gain from adding these features.

Store Polymorphism

As described in Section 1.4.3 store polymorphism is used to abstract the portion of the store that is of no concern for a given procedure. This is a convenient and useful abstraction mechanism. Without store polymorphism the type annotations have to describe the type of *all* locations in the store, and if dynamic allocation is allowed this is not even possible. Even when there is no dynamic allocation, having to is write down the type for the whole store for each procedure is unworkable for handwritten code.

Store polymorphism is not included in the extended type system in Section 3.6. This is an omission from my side. Simply put, I forgot it and when I discovered my error it was not the time to try and fix it.

However, I firmly believe that it would not be a problem to extend the type system from Section 3.6 to include store polymorphism and to extend the implementation to handle this feature. The mechanisms needed are similar to what is used for stack polymorphism.

Existential Types

In the current type system existential quantification is only allowed over index variables. In Section 4.1.4 and Section 4.2.1 we saw that existential quantification over at least location variables as well is needed for describing an array of arrays or pointer structures in general.

Aside for the example in Section 4.1.4 it is not clear how much gain general existential types would be for typical signal processing code which typically does not use pointer structures. Furthermore, if we introducing existential quantification over pointer variables we have to be careful or we will introduce type unsoundness. For example, given the following store type and regfile type:

```
XMEM[ p1 -> (? p . XMEM[ p -> fix [n]] xptr(p, 0)) [m] ]
[ i0 : xptr(p1, 0) ]
```

(where the question mark is used as ASCII notation for existential quantification). If we read an element from `i0`:

```
i1 = xmem[i0]
```

it is not enough to just update the regfile type with the new type for `i1`. That is, the following store type and regfile type is *not* correct:

```
XMEM[ p1 -> (? p . XMEM[ p -> fix [n]] xptr(p, 0)) [m] ]
[ i0 : xptr(p1, 0),
  i1 : ? p . XMEM[ p -> fix [n]] xptr(p, 0) ]
```

because these types does not record that the pointer in the first element of `p1` and the pointer in `i1` both point to the same location. Thus, when we read an element from `i0` we must change both the the store type and the regfile type to track alias information:

```
XMEM[ p' -> fix[n]
      p1 -> xptr(p', 0)[1]
          @ (? p . XMEM[ p -> fix [n]] xptr(p, 0)) [m-1] ]
[ i0 : xptr(p1, 0),
  i1 : xptr(p', 0) ]
```

Grossman [20] makes similar observations about the subtle interaction of mutation, aliasing, and existential types. Grossman shows how existential types can integrete in a C-like programming language, perhaps it is possible to adapt his techniques for Featherweight DSP.

May-Alias Types

One of the limitations of alias types is that it is impossible to describe that two pointers *may alias*, it is only possible to specify that two pointer are the same (by using the same location ρ) or that they are different (by using different location

variables ρ_1 and ρ_2). This is crucial for the destructive type-changing updates to be safe. This precision might result in inabilty to reuse a piece code that takes multiple arguments, where it does not matter whether the arguments alias or not. For example, the type annotations for the procedure `vecpmult_prefetch` on page 76 specify that `i0` and `i7` contain pointers to two district blocks of memory. But the code would work even if `i0` and `i7` contain the same pointer.

This limitation of the type system is inherited from Walker and Morrisett [52]. Smith et al. [50] describe how may-alias constraints can be added to a type system similar to mine. What is needed is that pointers that might be aliased must point to type-invariant aggregate objects. That is, it is possible to view the `array` types from the baseline type system as may-alias types, the problem is how to safely convert a `ptr` type to an `array` type.

Recursive Types

Walker and Morrisett [52] and Walker [53] have a μ operator for describing recursive types such as singly-linked lists, and a `rec` operator for describing parameterised recursive types such as doubly-linked lists or trees where the nodes have a parent pointer.

I think that it would not be a big problem to extend the type system and the implementation with these operators. Although one complication might be that these recursion operators are usually used together with existential quantification over location variables, and as we saw above we have to look out for the subtle interaction of mutation, aliasing, and existential types. Besides, pointer data structures are rare in embedded signal processing, so it is not clear that the extra complication is worthwhile in this context.

Sum Types

Experience from high-level programming languages shows that *sum types* (also known as *union types*) can be useful, especially for programs that do symbolic manipulations, such as compilers.

Xi and Harper [56] suggest to extend DTAL with sum types using the syntax:

$$\texttt{choose}(e, \tau_0, \ldots, \tau_{n-1})$$

where e is an index expression. This stands for a type which must be one of $\tau_0, \ldots, \tau_{n-1}$ determined by e: the type is τ_i if $e = i$.

As mentioned in Section 3.5 if we have sum types then it is not necessary to change the rule for `do`-loops to handle the prefetch idiom. It suffice to change the rule for reading from memory. Figure 6.1 shows the adapted rule for reading from memory using a `choose` type. In this rule we need to introduce a new index variable t, thus the judgement for small instructions need to be changed

$$\textbf{(read-choose)}\quad \frac{\begin{array}{c} R(r_2) = \tau\ \texttt{xarray}(e) \qquad t \notin \mathbf{dom}(\phi) \\ \phi' = \phi \wedge \{t : \texttt{int} \mid 0 \leq t \leq n \wedge (e > 0 \Rightarrow t = 0)\} \end{array}}{\Delta; \phi; \Psi; R \vdash r_1\ \texttt{=}\ \texttt{xmem[}r_2\texttt{]} \Rightarrow \phi'; R\{r_1 : \texttt{choose}(t, \tau, \texttt{junk})\}}$$

Figure 6.1: A type rule for reading from memory using `choose` types.

such that the index context is threaded through the rules. Instead of introducing a new index variable, we can use an alternative syntax for `choose` types:

$$\texttt{choose}(P_1 \Rightarrow \tau_1, \ldots, P_n => \tau_n)$$

This stands for a type which must be one of $\tau_1, \ldots, \tau_n$ determined by which P_i is true: the type is τ_i if P_i is true. The problem with this syntax is that we must ensure that if P_i and P_j both are true then τ_i and τ_j must be equal.

Aside from the reading from out of bounds I have not found any signal processing examples where I needed sum types, which is why they have been left out.

Pointer Equality

It would be nice if the type system could handle discovery of pointer equality. For example, if we add an extra branch instruction, `bpeq` to test equality of pointers:

$$\texttt{bpeq}\ r_1\texttt{,}\ r_2\texttt{,}\ v$$

This instruction transfers control to v if r_1 and r_2 are equal; otherwise execution continues with the following instruction and we know that r_1 and r_2 are not equal.

The type rule for the `bpeq` instruction would be something like (here for locations in X memory):

$$\frac{\begin{array}{c} R(r_1) = \texttt{xptr}(\rho_1, e_1) \\ R(r_2) = \texttt{xptr}(\rho_2, e_2) \\ \Psi; R \vdash v : \ \forall \Delta'. \forall \phi'. (\Psi', R') \\ \phi \wedge e_1 = e_2 \vdash \theta : \phi' \qquad \Delta_? ; \phi \wedge e_1 = e_2 \vdash \Theta : \Delta' \\ \Delta_? ; \phi \wedge e_1 = e_2 \models R_{???} <: R'[\Theta][\theta] \\ \Delta_? ; \phi \wedge e_1 = e_2 \models \Psi_{???} <: \Psi'[\Theta][\theta] \end{array}}{\Delta; \phi; \Psi; R \vdash \texttt{bpeq}\ r_1\texttt{,}\ r_2\texttt{,}\ v \Rightarrow \phi \wedge ?; \Psi_?; R_?}$$

But there are some problems that must be solved:

- What should the store type $\Psi_{???}$ and the regfile type $R_{???}$ be? In $\Psi_{???}$ and $R_{???}$ the locations ρ_1 and ρ_2 should be merged, and the types that they map to should be unified to the most specific. Also if one of the locations ρ_1 or ρ_2 is a location variable it should be removed from $\Delta_?$, and if both ρ_1 or ρ_2 are variables one of them should be removed.

- How should the index context be updated after the instruction? The problem is that the two pointers can be different for two reasons: either ρ_1 and ρ_2 are different or ρ_1 and ρ_2 are equal but e_1 and e_2 are not equal.

One way to work around these problems would be to restrict which pointers that can be compared. Again, we can take a leaf from the C standard [27] which specifies that only pointers into the same aggregate object (i.e., struct or array) can be compared (if two pointers that point into different aggregate objects are compared the result is unspecified). A type rule that enforces this restriction can easily be formulated:

$$\frac{\begin{array}{c} R(r_1) = \texttt{xptr}(\rho_1, e_1) \\ R(r_2) = \texttt{xptr}(\rho_1, e_2) \\ \rho_1 = \rho_2 \\ \Psi; R \vdash v : \ \forall \Delta'.\forall \phi'.\, (\Psi', R') \\ \phi \wedge e_1 = e_2 \vdash \theta : \phi' \qquad \Delta; \phi \wedge e_1 = e_2 \vdash \Theta : \Delta' \\ \Delta; \phi \wedge e_1 = e_2 \models R <: R'[\Theta][\theta] \\ \Delta; \phi \wedge e_1 = e_2 \models \Psi <: \Psi'[\Theta][\theta] \end{array}}{\Delta; \phi; \Psi; R \vdash \texttt{bpeq}\ r_1, \ r_2, \ v \Rightarrow \phi \wedge e_1 \neq e_2; \Psi; R}$$

By requiring that ρ_1 is equal to ρ_2 this rule enforces that the pointers are into the same aggrgate object

Nested Aggregate Types

In Section 3.6 I briefly mentioned that aggregate types are not as first-class as tuples normally are because aggregate types cannot be nested. That is, aggregate types are not allowed as element types for segments. The reason for this restriction is that if we allowed nested aggregate type, we could write the type of a flattened matrix with m columns and n rows as $(\tau[m])[n]$ (where m and n are index variables) which is equivalent to the flat aggregate type $\tau[m \cdot n]$, and this type use an index expression with multiplication of variables which is not allowed in Presburger arithmetic.

But we could allow nested aggregate types with at most one index variable involved in the nesting. That is, types like $(\tau[2])[n]$ or $(\tau[n])[2]$ which are both equivalent to the flat aggregate type $\tau[2 \cdot n]$. This would enable succinct descriptions of aggregate type such as $(\texttt{int}(e)[1]\texttt{@fix}[1])[64]$ which are currently unworkable cumbersome to write out.

Extending the syntax to allow nested aggregate types with at most one index variable involved in the nesting would be easy, and because these types can be written as flat aggregate types it can be viewed as a pure preprocessing step. Thus, none of the current typing rules or the implementation (except for the parser) would have to changed.

(**read-pd**)
$$\frac{R(r_2) = \tau\ \texttt{xarray}(e)\{i\} \qquad \phi \models e > 0}{\Delta; \phi; \Psi; R \vdash r_1\ \texttt{=}\ \texttt{xmem[}r_2\texttt{]} \Rightarrow [r_1 : \tau[i \mapsto 0]]}$$

(**write-pd**)
$$\frac{R(r_1) = \tau_1\ \texttt{xarray}(e)\{i\} \qquad \phi \models e > 0 \qquad R(r_2) = \tau_2 \qquad \Delta; \phi \models \tau_2 <: \tau_1[i \mapsto 0]}{\Delta; \phi; \Psi; R \vdash \texttt{xmem[}r_1\texttt{]}\ \texttt{=}\ r_2 \Rightarrow []}$$

(**incr-pd**)
$$\frac{R(r) = \tau\ \texttt{xarray}(e_1)\{i\} \qquad \Psi; R \vdash \mathit{aexp} : \texttt{int}(e_2) \qquad \phi \models e_2 > 0}{\Delta; \phi; \Psi; R \vdash r\ \texttt{+=}\ \mathit{aexp} \Rightarrow [r : \tau[i \mapsto i + e_2]\ \texttt{xarray}(e_1 - e_2)\{i\}]}$$

Figure 6.2: Type rules for position dependent types.

Position Dependent Types

It would be interesting to try and extend the type system to allow arrays where the type of an element is dependent on the position of the element in the array. A possible syntax for this could be:

$$\tau\ \texttt{xarray}(e)\{i\}$$

where i is an index variable which is bound in τ and we know that $0 \leq i < e$. That is, the rule for well-formed arrays must be changed to:

$$\frac{i \notin \mathbf{dom}(\phi) \qquad \Delta; \phi \wedge \{i : \texttt{int} \mid 0 \leq i < e\} \vdash_{\text{wf}} \tau \qquad \phi \vdash_{\text{wf}} e}{\Delta; \phi \vdash_{\text{wf}} \tau\ \texttt{xarray}(e)\{i\}}$$

Position dependent types would allow us to, for example, write the type of an integer array with n elements where the element at position i has the value i:

$$\texttt{int}(i)\ \texttt{xarray}(n)\{i\}$$

And we can also write the type of an integer array with n elements where the element at position i has a value e which is strictly smaller than i:

$$(\exists\{e : \texttt{int} \mid e < i\}.\texttt{int}(e))\ \texttt{xarray}(n)\{i\}$$

The hard part of this extension is how to adapt the rules for pointer arithmetic, reading from memory, and writing to memory where we have to be a bit careful. Figure 6.2 shows rules for incrementing an array pointer, for reading from memory, and for writing to memory. These rules use substitutions to ensure the position variable i does not escape its scope. The crucial rule is the rule (**incr-pd**) for incrementing a pointer: when a register r that contains a pointer to an array with e_1 elements, is incremented with e_2, then r contains a pointer to an array with $e_1 - e_2$ elements. The element that used to be at position i with type τ (which may contain the index variable i) is now at position $i - e_2$, but we

have not changed the element, thus the type of the element must now be τ with all occurrences of i replaced with $i + e_2$.

I have not found any direct use for this extension alone for signal processing algorithms. The original motivation for position dependent types was to combine it with some form of sum types to handle initialisation and reuse of arrays, but I was never able to work out all the details and instead I turned to alias types which I think is a much nicer solution.

It is also possible to extend aggregate types with position dependent types. The first thing we have to decide is, whether the type of an element should be dependent on the position of the element in the current segment, or the position of the element in the whole aggregate object. It is also hard to find a syntax that fits well with the current syntax for aggregate types. Given that I do not have a really useful example on which to test this extension, I have not worked out the details for aggregate types.

Mix Existential Quantification and Aggregate Types

It would be nice to allow existential quantification of index variables over aggregate types. This would for example allow us to specify that at location ρ in X memory is a sorted integer array of size n:

$$\begin{array}{l}\{n : \texttt{int}, i : \texttt{int} \mid 0 \leq i < n\} \\ \texttt{XMEM}[\rho \mapsto \exists\{f : \texttt{int}\}.(\quad (\exists\{e : \texttt{int} \mid e < f\}.\texttt{int}(e))[i] \\ \qquad\qquad\qquad\qquad\quad @ \quad \texttt{int}(f)[1] \\ \qquad\qquad\qquad\qquad\quad @ \quad (\exists\{g : \texttt{int} \mid f < g\}.\texttt{int}(g))[n - i - 1])\end{array}$$

What this type says is that, for all positions i between zero and n there exists an integer f such that the aggregate at ρ in X memory can be split into three segments. First, there is a segment with i elements, all integers strictly less than f (but the elements are not necessarily equal to each other):

$$(\exists\{e : \texttt{int} \mid e < f\}.\texttt{int}(e))[i]$$

Second, there is a segment of one element that contains the integer f:

$$\texttt{int}(f)[1]$$

Finally, there is a segment with $n - i - 1$ elements, all integers strictly greater than f (but the elements are not necessarily the equal to each other):

$$(\exists\{g : \texttt{int} \mid f < g\}.\texttt{int}(g))[n - i - 1]$$

The only way these contraints can be satisfied is if the array at ρ is sorted and all the elements are different.

While it is straightforward to extend the type syntax to allow existential quantification of index variables over aggregate types and to extend the algorithm from Section 5.3.2 to decide subtyping and equality of such aggregate types, it is not clear how to adapt the rules in Figure 3.21 for reading from and writing to memory.

Soundness Proof

The main purpose of the work presented in this thesis has been to test whether it is possible to make a nice type system for low-level handwritten assembler code. Hence, the focus has been on the design of the type system. Before we knew that the type system presented is practically useful the marginal utility of making a formal proof was small. Now when the work is more complete and the design has settled a bit it is worth to try and complete a formal soundness proof for the presented type system.

6.1.3 Systematic Testing

As described in the previous section, we can make a formal proof to validate the type rules. And while the might raise our confidence for the type system, it will say nothing about whether the implementation actually implements the type system described by the typing rules. We might consider making a formal proof of correctness and correspondance for the implemtation, but the type systems described in this dissertation are rather compilcated and the implementation involve a complex decision procedure for Presburger arithmetic. Thus, a complete formal proof is probably not feasible with todays techniques.

The next best thing to a formal proof is systematic testing. To systemtically test whether an implementation of a type checker correspond to a set of typing rules we might build a test suite, where there is at least two test for each typing rule: one where the rule succeeds and one where it fails. Futhermore, for each typing rule with more that one premise we can make tests such each premise is tested. For example, for the rule (**beq**) in Figure 3.9 on page 47 we would need at least six tests. Five failure tests: one test where r is not an integer, one test where v is not a code address, one test where we cannot jump to v because we cannot find a substitution for the index variables, one test where we cannot find a substitution for the type and stack variables, and one test where the regfile type R is not a subtype of $R'[\Theta][\theta]$; and one test where all premises are satisfied and the rule succeeds.

This strategy will work well for the (mostly) syntax directed judgements. But for the more complex parts of the type systems in Chapter 3, such as subtype checking for aggregate types, we will need a sighly different strategy to test interesting corner cases. For example, we want failure and success test cases for each of the six cases in Figure 5.3 plus tests for the limits of these cases.

While such a systematic and rigorous test suite would raise our confidence in the implementation (and possible also in the type system itself), it is of course no guarantee for correctness.

6.1.4 Larger Examples

The examples in the benchmark suite from Section 5.5 gives a good indication of how the type annotations can be used for handwritten DSP code and how

well the implementation performs on such programs, it would be nice with some more examples and perhaps also some larger examples as well. One way to make the benchmark suite more convincing would be to port all the ROM primitives from the industrial partner's code to Featherweight DSP. The biggest problem of doing so would be how to handle the features Featherweight DSP have left out, see Section 6.1.1. If Featherweight DSP, the type system, and the implementation were extended with just the ability to handle reading from and writing to peripheral space, and reading from and writing to the same address in both X and Y memory, then most of the ROM primitives could be translated real custom DSP assembler to Featherweight DSP automatic.

6.1.5 Improve Implementation

In this section I list some of the improvements to my proof-of-concept implementation that I would like to implement, but have yet to find time for.

Proper Quantifier Elimination

As described in Section 5.4.1 I have only implemented at crude incomplete form of quantifier elimination. There exists a complete algorithm for quantifier elemination described by Pugh and Wonnacott [45] and this algorithm is implemented by Norrish in the Kananaskis release (and later) of the theorem prover HOL4 [37]. Hence, I do not think that it would be too much work to add proper quantifier elimination to my implementation.

Better Handling of Equality Constraints

In my current implementation, equality constraints, that is, index propositions on the form $e_1 = e_2$, are used only for quantifier elimination. After quantifier elimination, equality constraints are rewritten to two inequalities $e_1 \leq e_2 \wedge e_2 \leq e_1$ which is suboptimal.

Pugh and Wonnacott [45] state that it is vital for the performance on their example to first eliminate as many variables as possible by rewriting with equalities and Norrish [38] reports similar experience. The performance of my simple implementation has been more that adequate for my experiments, but a better one would permit more convenient handling of formulae such as:

$$m = 0 \wedge n * m \leq 0$$

(where m and n are variables). At first sight this is not a Presburger formula, because the variables m and n are multiplied. But if we rewrite with the first equality, then we get the formula:

$$n * 0 \leq 0$$

which is a Presburger formula (and is true). Such preprocessing of the formulae would be useful for procedures which only need to work for a finite set of constants.

Automatic Elimination of Division and Modulo Operations With Constants

Formulae with divisions and modulo operations with constant arguments can be automatically rewritten to formulae without division and modulo. Again, this is implemented in HOL4 so it should not be difficult to add to my implementation.

Together with a better handling of equality constraints this would enable the implementation to handle the modulo auto-increment addressing mode (see Section 2.1.5) of the real custom DSP.

Alternative Data Structure For Managing Constrains

As described in Section 5.4, the current implementation expands a formula into Disjunctive Normal Form (DNF) before feeding the formula piecewise, one disjunct at a time, to the core engine. Remember that the core engine can be used to find satisfying assignments to formulae on core form (see (**core**) on page 100). The expansion to DNF results in an exponential blowup of the formula, which is undesirable.

Instead of expanding to DNF we could use Binary Decision Diagrams (BDDs) [4] to manage what is fed to the core engine: Given a formula P, to each distinct subformula C_i on the form:

$$0 \leq c_{i1}x_1 + c_{i2}x_2 + \cdots + c_{in}x_n$$

(that is, each leaf) assign a Boolean BDD variable b_i, then build the BDD B for the Boolean formula P', where P' is $P[C_i \mapsto b_i]$ for all C_i in P. We now traverse all paths from the root of B to the terminal node **1** one path at a time, translate each path to a formula on core form, and feed this formula to the core engine until a contradiction is found. A path is translated to a formula by translating each node on the path to a constraint and then make a conjunction of all these constraints. To translate a node n_i to a constraint we first find the associated BDD variable b_i, then we find the constraint C_i that b_i was assigned to, and if we follow the high edge (*true*) from n_i in the path we are translating then C_i is the translation of n_i; otherwise, if we follow the low edge (*false*), $\neg C_i$ is the translation of n_i.

There is no guarantee that B will not be exponentially larger than the original formula P, but from model checking we know that B is often propotional to P. But even if B is propotional to P there might still be an exponential number of paths to follow in B. Still, this algorithm seems to offer many more opportunities for sharing and optimisations than using DNF.

Chan et al. [7], Seshia and Bryant [49] use similar techniques for combining BDDs and Presburger arithmetic.

Another possibility would be to try and extend BDDs such that they can represent Presburger formulae. Difference Decision Diagrams (DDDs) [32] are an example on how to extend BDDs to represent a first order logic over constraints

of the form $x - y \leq d$, where x and y are variables and d is a constant. Linear Decision Diagrams (LDDs) [19] can represent Presburger formulae using a BDD-like data structure.

Local Type Inference

In Chapter 4 and Section 5.5 we saw that with the current implementation it is necessary with a somewhat large amount of type annotation in the source code. I think that many of these annotations could be automatically inferred, because most of them I have inferred mechanically by hand. The annotations for `do`-loop looks like they are a good place to start, as they usually just fall out from the context and the body of the loop.

The problem with type inference is that there are no principal typings for indexed types, which means that in general no best type can be inferred.

6.2 Related Work

This section compare my work with related research.

6.2.1 Typed Assembler Languages

Since the influential work of Morrisett et al. [35] there has been a lot of research of type systems for low-level languages. Figure 6.3 shows a schematic comparison of Featherweight DSP and five other typed assembly languages. The other languages in Figure 6.3 are:

- TAL. Based on the papers [33, 34, 35, 50, 52, 21] which describe different subset of the implementation TALx86 for the IA32 instruction set architecture.
- DTAL. Based on [56, 55].
- LowTAL. Low-level Typed Assembly Language, based on [8]. The authors use the name LTAL but this creates a name conflict in this comparison. LowTAL is based on Typed Machine Language [51]. It can be translated to real Sparc code.
- LinTAL. Based on [9]. Again, the original name used by the authors is LTAL. LinTAL also allow direct manual reuse of memory, in some sense their `cell` type corresponds to my `junk` type: one word of memory. But they do not have indexed types.
- TALT. TAL Two, based on [12]. Like LowTAL TALT is intended to be a fully formalised and have machine-checkable safety proof.

The categories measured in Figure 6.3 are:

Legend: ● yes ○ some no	Bounds check elimination	Control-flow sensitive	General pointer arithmetic	Relies on garbage collector	Pseudo-instructions	Build-in data type repr.	Explicit memory reuse	Real Machine	Machine checked proofs
TAL		○		●	○		○	●	
DTAL	●	●		●	○	●			
LinTAL		○				●	●		
LowTAL	○	●	○	●			○	●	●
TALT	○	●	○	●	○		○	●	●
Featherweight DSP	●	●	○				●	○	

Figure 6.3: Comparison of different type system for low-level languages.

Bounds check elimination. Does the type system track the value of integer exressions so that bounds check for array indexing can be moved around freely and potentially be completely eliminated. Only DTAL and Featherweight DSP offer full support. LowTAL and TALT does track some limited form of information about integer expressions, which potentially could be used for eliminating bound check.

Control-flow sensitive. All the type system support either alias types or sigleton types of some sort, which enables the type system to be control-flow sensitive.

General pointer arithmetic. Only LowTAL, TALT, and Featherweight DSP support pointer arithmetic. None support arbitrary arithmetric operations on pointers. The focus for pointer arithmetic is different, Featherweight DSP concentrates on supporting pointer arithmetic to enable explicit and flexible data manipulation and reuse, whereas LowTAL and TALT concentrates on pointer arithmetic for code addresses. LowTAL supports enough just pointer arithmetic to enable position-independent code. TALT support enough pointer arithmetic to enable relative addressing.

Relies on garbage collector. Only LinTAL and Featherweight DSP have the explicit goal of *not* relying on a garbage collector. But LinTAL have support for co-existing with a garbage collector. Featherweight DSP does not even support dynamic allocation of memory.

Pseudo-instructions. Some TALs have pseudo-instructions for compare-and-branch, datatype tag-checking, or memory allocation which is translated to a sequence of real machine instructions or a call to a runtime system (this is called *atomicity* in [8]). For DTAL and LinTAL this is hard to determine because they do not compile to real machine code, the basis for the markings in this coloumn is that DTAL has an `alloc` which cannot be implemented with a single instruction on conventional hardware. LinTAL on the other hand does not have an `alloc` instruction. The other markings in this coloumn is based on the table in [8].

Build-in data type repr. Does the system comes with predefined datatypes such as arrays and tuples, or can the user (human or compiler) of the system freely chose data type representation. DTAL comes with predefined arrays, and the only datatype in LinTAL is pairs.

Explicit memory reuse. Can memory be explicitly managed and reused or does the system enforce the type invariance principle. The only two system that has explicit memory management clearly stated as a goal is LinTAL and Featherweight DSP. TAL, LowTAL, and TALT seems to be using alias types, but only for seperating allocation and seperation initialisation.

Real Machine. Does the system support tranlation to a real machine language. TAL and TALT is based Pentium code and each LowTAL instruction corresponds to at most one Sparc instruction. DTAL and LinTAL only have interpreters for their instruction sets. Featherweight DSP can mostly be translated straightforwardly to real custom DSP assembler, the only exception is the branching instructions.

Machine checked proofs. TAL, DTAL, and LinTAL only have hand-checked proof. LowTAL and TALT have machine checked proofs (which are mostly done). Featherweight DSP has no formal proof at all.

The languages compared in Figure 6.3 are by far not the only related research on type system for low-level languages, others include Heap Bounded Assembly Language [3] and Crary and Weirich [13]. Both of these use type systems to give upper limit guarantees on resource bounds.

Ahmed and Walker [1] presents a logical framework for reasoning about adjancency, seperation of memory blocks, and aliasing. The logic is deplyed as a type system for a formal model stack-based assembly language. Their modality for adjancency is similar to my append operator `@`for aggregate types. But their modalities are not combined with index types.

6.2.2 Sized Types

The work on Sized Types by Pareto [41] and Chin et al. [10] is similar to my work. We both employ type systems based on Presburger arithmetich to trach

the size of arrays and to find certain classes of errors in software for embedded systems. While Pareto devise type systems that can catch an impressive array of classes of program errors:

- non-exhaustive patterns,
- partial numerical operators,
- partial library functions,
- explicit failures,
- call chains too large for the stack,
- data-structures too large for the heap,
- space leaks,
- rate conflicts,
- busy loops,
- deadlocks.

Pareto does so by introducing two new programming languages (neither can handle the full list of classes of errors) with associated programming styles. I, in contrast, concentrate on a much smaller list of classes of errors, and try to follow the guidelines from Section 1.1.4. That is, I work with the existing coding style and an existing assembler language used in embedded systems today.

6.2.3 Cyclone

As described in Section 1.4.4, Cyclone [28, 22] shares many goals with the work presented in this dissertation. Cyclone is a low-level language with a high-level type system, and Cyclone targets handwritten code. But Cyclone concentrates on security and not resource contrained embedded software, thus some trade-offs have been made which are not appropriate for embedded DSP code. Cyclone has for example support for several flavours of dynamic memory allocation: allocation in a global heap which is garbage collected, stack allocation corresponding to local-declaration blocks in C, and dynamically growable regions. But Cyclone does not support manual management of static memory where a static location in memory can contain objects of different types at different points in time during execution.

I think that my work and Cyclone could be interesting to combine. This combination could be taken in two directions:

- To use Cyclone to widen the scope of my work. While I have concentrated on code for embedded DSP programs, I believe that my combination of alias types and indexed types, and aggregate types are useful outside the scope of embedded DSP code. An interesting project would be to extend Cyclone with these constructs.

  Jim et al. [28], for example, reports that the biggest performance problem of Cyclone stems from the use of *fat pointers*.

My combination of alias types and indexed types can be seen as fat pointers checked at compile time rather than at runtime, and thus bringing no runtime overhead, neither in execution time nor memory for storing bounds.

- To extend Cyclone with features useful for embedded DSP code. Similar to how DSP-C [15] extends C. That is, Cyclone could be extended with language support for fixed-point number and circular arrays and pointers so that signal processing algorithms can be efficiently compiled to DSPs.

6.2.4 AnnoDomini

Eidorff et al. [16] and Ramalingam et al. [46] use a type system and type inference to detect and repair Year 2000 problems in COBOL programs. Their type construct for COBOL records are similar to my aggregate types. Like my aggregate types their record types allow different views. But their record type have statically known sizes whereas my aggregate types can have a symbolic size which might be unknown.

Chapter 7

Conclusion

7.1 Summary

In this dissertation I have presented my thesis:

> *A high-level type system is a good aid for developing signal processing programs in handwritten Digital Signal Processor (DSP) assembler code.*

To test this thesis I have made a model assembler language called Featherweight DSP which captures some of the essential features of a real custom DSP used in the industrial partner's digital hearing aids: zero-overhead looping hardware, instruction-level parallelism, hardware support for procedure abstraction, and sequential traversal of arrays using pointer arithmetic. I have presented a baseline type system which is the type system of DTAL adapted to Featherweight DSP. Then I have explained two classes of programs that uncovers some shortcomings of the baseline type systesm. The classes of problematic programs are exemplified by a procedure that initialises an array for reuse, and a procedure that computes point-wise vector multiplication. The latter uses a common idiom of prefetching memory resulting in out-of-bounds reading from memory. I then present two extensions to the baseline type system: The first extension is a simple modification of some type rules to allow out-of-bounds reading from memory. The second extension is based on two major modifications of the baseline type system:

- Abandoning the type-invariance principle of memory locations and using a variation of alias types instead.
- Introducing aggregate types making it possible to have different views of a block of memory, thus enabling type checking of programs that directly manage and reuse memory.

I then show that both the baseline type system and the extended type system can be used to give type annotations to handwritten DSP assembler code, and that these annotations precisely and succinctly describe the requirements of

a procedure. I have implemented a combined proof-of-concept type checker for both the baseline type system and the extended type system. I get good performance results on a small benchmark suite of programs representative of handwritten DSP assembler code. The good performance is achieved despite that I have used a simpleminded implementation strategy and use Moscow ML (which is an interpreter) as implementation platform. These empirical results are encouraging and strongly suggest that it is possible to build a robust implementation of the type checker which is fast enough to be called every time the compiler is called, and thus can be an integrated part of the development process.

7.2 Contributions

With this dissertation I have made the following main contributions:

- I have introduced a small well-defined formal model assembler language called Featherweight DSP. Even though Featherweight DSP is derived from the assembler language for a custom DSP, Featherweight DSP captures the essential features generally found for embedded fixed-point DSPs.

  Thus, Featherweight DSP can be used as a stepping stone into the field of embedded signal processing programs for researchers in programming language theory who are interested in this field, but do not have the time for a full scale domain investigation.

- I have successfully adapted the type system of DTAL to Featherweight DSP, and shown how to handle the features of embedded DSPs with this type system.

  This enables us to show that an unchanged DTAL type system is not suitable for handwritten assembler code. DTAL is designed to be a machine-generated target language and relies on a runtime system. Thus, programs in DTAL are not meant to directly manage and reuse memory.

- I have shown how alias types and indexed types can be combined to handle restricted pointer arithmetic.

- I have introduced a novel type construct called aggregate types. Aggregate types allow different views on a block of memory, and are flexible enough to allow direct memory reuse in programs.

- I have implemented a proof-of-concept type checker and clearly described the limitation of the implementation. Despite that the theoretical complexity of the type checker is worse that super-exponential [40] in the size of checked annotations, a small empirical study suggests that it practically feasible to use the type checker in the daily development process.

All in all I believe that I have shown that it is possible to develop a practically useful type checker for handwritten DSP assembler code with type annotations; that such a type checker can help to catch certain classes of untrapped errors, such as memory safety violations; and that the type annotations also will supplement the documentation of assembler code.

While my work has been narrowly concentrated on code for embedded DSP programs, I firmly believe that my combination of alias types and indexed types, and aggregate types are useful outside this scope.

The question is: what is a Mahnamahna?

Appendix A

Complete Example Code Listings

A.1 Fill an array with zeros

The complete listing of the example described in Section 4.1.2, Figure 4.3 with no types elided.

```
1   fill_zero:
2       (p, r, s)
3       {n : int | n > 0}
4       XMEM[ p -> junk[n] ]
5       [ i0  : xptr(p, 0)
6       , i11 : int(n)
7       , dsp : r
8       , csp : XMEM[ p -> int(0)[n] ]
9               [ x0  : junk
10              , i0  : xptr(p, n)
11              , i11 : int(n)
12              , dsp : r
13              , csp : s
14              ] :: s
15      ]
16      x0 = 0
17      do (i11) {
18         (p, r, s)
19         {n : int, k : int | n > 0 /\ 0 <= k < n}
20         XMEM[ p -> int(0)[k] @ junk[n-k] ]
21         [ x0  : int(0)
22         , i0  : xptr(p, k)
23         , i11 : int(n)
24         , dsp : int(k) :: r
25         , csp : XMEM[p -> int(0)[n]]
26                 [ x0  : junk
```

```
                , i0  : xptr(p, n)
                , i11 : int(n)
                , dsp : r
                , csp : s
                ] :: s
        ]
        xmem[i0] = x0; i0 += 1
    }
    ret
```

A.2 Pointwise Vector Multiplication with Prefetch

The complete listing of the example described in Section 4.1.3, Figure 4.4 with no types elided.

```
vecpmult_prefetch:
    (s, r, p1, p2, p3)
    {n : int  |  n > 1}
    XMEM[ p1 -> fix[n], p3 -> junk[n] ]
    YMEM[ p2 -> fix[n] ]
    [ i0  : xptr(p1, 0),
      i4  : yptr(p2, 0),
      i1  : xptr(p3, 0),
      i7  : int(n-1),
      dsp : r,
      csp : XMEM[ p1 -> fix[n], p3 -> fix[n] ]
            YMEM[ p2 -> fix[n] ]
            [ x0  : junk, y0 : junk, a0 : junk,
              i0  : xptr(p1, n+1),
              i4  : yptr(p2, n+1),
              i1  : xptr(p3, n),
              i7  : int(n),
              dsp : r,
              csp : s
            ] :: s
    ]
    x0  = xmem[i0]; i0+=1
    y0  = ymem[i4]; i4+=1
    do (i7) {
       (s, r, p1, p2, p3)
       {n : int, k : int | n > 1 /\ 0 <= k < n}
       XMEM[ p1 -> fix[n], p3 -> fix[k] @ junk[n-k] ]
       YMEM[ p2 -> fix[n] ]
       [ x0  : fix,
```

```
            y0  : fix,
            i0  : xptr(p1, k+1),
            i4  : yptr(p2, k+1),
            i1  : xptr(p3, k),
            i7  : int(n),
            dsp : int(k) :: r,
            csp : XMEM[ p1 -> fix[n], p3 -> fix[n] ]
                  YMEM[ p2 -> fix[n] ]
                  [ x0  : junk, y0 : junk, a0 : junk,
                    i0  : xptr(p1, n+1),
                    i4  : yptr(p2, n+1),
                    i1  : xptr(p3, n),
                    i7  : int(n),
                    dsp : r,
                    csp : s
                  ] :: s
          ]
          a0=x0*y0; x0 = xmem[i0]; i0+=1; y0 = ymem[i4]; i4+=1
          xmem[i1] = a0; i1+=1
       }
       ret
```

A.3 Matrix Multiplication

This section lists the code for the matix multiplication example in described in Section 4.1.4 in two different versions that only differs in the type annotations.

A.3.1 Matrix Multiplication Baseline Types

The complete listing of the example described in Section 4.1.4, Figure 4.5 with no types elided.

```
matrix_mult:
   (s, r)
   {rows : int, col1 : int, col2 : int
         | rows > 0 /\ col1 > 0 /\ col2 > 0}
   [ i6  : int(rows),
     i7  : int(col1),
     i8  : int(col2),
     i0  : fix xarray(col1) xarray(rows),
     i1  : fix yarray(col2) yarray(col1),
     i2  : fix xarray(col2) xarray(rows),
     dsp : r,
     csp : [ i6  : int(rows),
```

```
13              i7  : int(col1),
14              i8  : int(col2),
15              i0  : fix xarray(col1) xarray(0),
16              i1  : fix yarray(col2) yarray(col1),
17              i2  : fix xarray(col2) xarray(0),
18              a0  : junk, x0 : junk, y0 : junk,
19              i3  : junk, i4 : junk, i5 : junk,
20              i10 : junk, i11 : junk, i12 : junk,
21              dsp : r,
22              csp : s] :: s
23      ]
24      do (i6) {
25        {rows : int, col1 : int, col2 : int,
26           k1 : int
27              | rows > 0 /\ col1 > 0 /\ col2 > 0
28             /\ 0 <= k1 < rows
29        }
30        [ i6  : int(rows),
31          i7  : int(col1),
32          i8  : int(col2),
33          i0  : fix xarray(col1) xarray(rows-k1),
34          i1  : fix yarray(col2) yarray(col1),
35          i2  : fix xarray(col2) xarray(rows-k1),
36          dsp : int(k1) :: r,
37          csp : [ i6  : int(rows),
38                  i7  : int(col1),
39                  i8  : int(col2),
40                  i0  : fix xarray(col1) xarray(0),
41                  i1  : fix yarray(col2) yarray(col1),
42                  i2  : fix xarray(col2) xarray(0),
43                  a0  : junk, x0 : junk, y0 : junk,
44                  i3  : junk, i4 : junk, i5 : junk,
45                  i10 : junk, i11 : junk, i12 : junk,
46                  dsp : r,
47                  csp : s] :: s
48        ]
49        i3  = xmem[i0]; i0 += 1
50        i5  = xmem[i2]; i2 += 1
51        i10 = 0
52        do (i8) {
53          {rows : int, col1 : int, col2 : int,
54             k1 : int, k2 : int
55                | rows > 0 /\ col1 > 0 /\ col2 > 0
56               /\ 0 <= k1 < rows
```

```
57              /\ 0 <= k2 < col2
58          }
59          [ i6  : int(rows),
60            i7  : int(col1),
61            i8  : int(col2),
62            i0  : fix xarray(col1) xarray(rows-k1),
63            i1  : fix yarray(col2) yarray(col1),
64            i2  : fix xarray(col2) xarray(rows-k1),
65            i3  : fix xarray(col1),
66            i5  : fix xarray(col2-k2),
67            i10 : int(k2),
68            dsp : int(k2) :: int(k1) :: r,
69            csp : [ i6  : int(rows),
70                    i7  : int(col1),
71                    i8  : int(col2),
72                    i0  : fix xarray(col1) xarray(0),
73                    i1  : fix yarray(col2) yarray(col1),
74                    i2  : fix xarray(col2) xarray(0),
75                    a0  : junk, x0 : junk, y0 : junk,
76                    i3  : junk, i4 : junk, i5 : junk,
77                    i10 : junk, i11 : junk, i12 : junk,
78                    dsp : r,
79                    csp : s] :: s
80          ]
81          a0 = 0
82          i11 = i1
83          i12 = i3
84          do (i7) {
85            {rows : int, col1 : int, col2 : int,
86               k1 : int, k2 : int, k3 : int
87                  | rows > 0 /\ col1 > 0 /\ col2 > 0
88                /\ 0 <= k1 < rows
89                /\ 0 <= k2 < col2
90                /\ 0 <= k3 < col1
91            }
92            [ i6  : int(rows),
93              i7  : int(col1),
94              i8  : int(col2),
95              i0  : fix xarray(col1) xarray(rows-k1),
96              i1  : fix yarray(col2) yarray(col1),
97              i2  : fix xarray(col2) xarray(rows-k1),
98              i3  : fix xarray(col1),
99              i5  : fix xarray(col2-k2),
100             i10 : int(k2),
```

```
101               a0  : fix,
102               i11 : fix yarray(col2) yarray(col1-k3),
103               i12 : fix xarray(col1-k3),
104               dsp : int(k3) :: int(k2) :: int(k1) :: r,
105               csp : [ i6  : int(rows),
106                       i7  : int(col1),
107                       i8  : int(col2),
108                       i0  : fix xarray(col1) xarray(0),
109                       i1  : fix yarray(col2) yarray(col1),
110                       i2  : fix xarray(col2) xarray(0),
111                       a0  : junk, x0 : junk, y0 : junk,
112                       i3  : junk, i4 : junk, i5 : junk,
113                       i10 : junk, i11 : junk, i12 : junk,
114                       dsp : r,
115                       csp : s] :: s
116             ]
117             i4 = ymem[i11]; i11 += 1
118             x0 = xmem[i12]; i12 += 1
119             i4 = i4 + i10
120             # nop instruction needed here in real custom DSP assembler
121             y0 = ymem[i4]
122             a0 += x0 * y0
123           }
124           xmem[i5] = a0; i5 += 1
125           i10 += 1
126         }
127       }
128       ret
```

A.3.2 Matrix Multiplication Extended Types

This version uses type annotations from the extended type system. Thus, each of the three matrices only contains one shared row, according to the type system.

Types have been elided in this version, because the implementation automatically insert the types that should be repeated.

```
1  matrix_mult:
2      (s, r, p1, p2, p3, p4, p5, p6)
3      {rows : int, col1 : int, col2 : int
4            | rows > 0 /\ col1 > 0 /\ col2 > 0}
5       XMEM[ p1 -> fix[col1],
6             p3 -> xptr(p1, 0)[rows],
7             p5 -> junk[col2],
```

```
8             p6 -> xptr(p5, 0)[rows]
9           ]
10       YMEM[ p2 -> fix[col2],
11             p4 -> yptr(p2, 0)[col1]
12          ]
13      [ i6  : int(rows), i7  : int(col1), i8  : int(col2),
14        i0  : xptr(p3, 0),
15        i1  : yptr(p4, 0),
16        i2  : xptr(p6, 0),
17        dsp : r,
18        csp : !(){}.
19              XMEM[ p1 -> fix[col1]
20                  , p3 -> xptr(p1, 0)[rows]
21                  , p5 -> fix[col2]
22                  , p6 -> xptr(p5, 0)[rows]
23                  ]
24              YMEM[ p2 -> fix[col2]
25                  , p4 -> yptr(p2, 0)[col1]
26                  ]
27              [ i6  : int(rows), i7 : int(col1), i8 : int(col2),
28                i0  : xptr(p3, rows),
29                i1  : yptr(p4, 0),
30                i2  : xptr(p6, rows),
31                a0  : junk, x0 : junk, y0 : junk,
32                i3  : junk, i4 : junk, i5 : junk,
33                i10 : junk, i11 : junk, i12 : junk,
34                dsp : r,
35                csp : s] :: s
36      ]
37      do (i6) {
38        { k1 : int | 0 <= k1 < rows }
39        [ i0  : xptr(p3, k1),
40          i2  : xptr(p6, k1),
41          dsp : int(k1) :: r
42        ]
43        i3  = xmem[i0]; i0 += 1
44        i5  = xmem[i2]; i2 += 1
45        i10 = 0
46        do (i8) {
47          { k2 : int | 0 <= k2 < col2 }
48          XMEM[ p5 -> fix[k2] @ junk[col2-k2] ]
49          [ i3  : xptr(p1, 0),
50            i5  : xptr(p5, k2),
51            i10 : int(k2),
```

```
52            dsp : int(k2) :: int(k1) :: r
53          ]
54          a0 = 0.0
55          i11 = i1
56          i12 = i3
57          do (i7) {
58            { k3 : int | 0 <= k3 < col1 }
59            [ a0  : fix,
60              i11 : yptr(p4, k3),
61              i12 : xptr(p1, k3),
62              dsp : int(k3) :: int(k2) :: int(k1) :: r
63            ]
64            i4 = ymem[i11]; i11 += 1
65            x0 = xmem[i12]; i12 += 1
66            i4 = i4 + i10
67            y0 = ymem[i4]
68            a0 += x0 * y0
69          }
70          xmem[i5] = a0; i5 += 1
71          i10 += 1
72        }
73      }
74      ret
```

A.4 Sum of Imaginary Parts

Takes an array of complex numbers represented as an flat array of fix point numbers as input, and sums all the imaginary parts, that is, all the elements on the odd indexes. The code uses the prefetch idiom.

The code is listed in two different versions that only differs in the type annotations.

A.4.1 Baseline Types

The type annotations in this version are drawn from the baseline type system.

Types have been elided in this version, because the implementation automatically insert the types that should be repeated.

```
1  add_im_part_prefetch:
2       (s, r)
3       { n : int |  n > 0}
4       [ i0  : fix xarray(2*n),
5         i7  : int(n),
6         dsp : r,
```

```
        csp : [ a0  : fix,
                i0  : fix xarray(-2),
                dsp : r,
                csp : s
              ] :: s
    ]
    a0 = 0.0
    x0 = xmem[i0]; i0+=2
    do (i7) {
       { k : int  | 0 <= k < n}
       [ a0  : fix,
         x0  : fix,
         i0  : fix xarray(2*n - 2*k - 2),
         dsp : int(k) :: r
       ]
       a0 += x0; x0 = xmem[i0]; i0+=2
    }
    ret
```

A.4.2 Extended Types

The type annotations in this version are drawn from the extended type system.

Types have been elided in this version, because the implementation automatically insert the types that should be repeated.

```
add_im_part_extended:
    (s, r, p)
    { n : int |  n > 0}
    XMEM[p -> fix[n + n]
    [ i0  : xptr(p, 0),
      i7  : int(n),
      dsp : r,
      csp : XMEM[p -> fix[n] @ fix[n] ]
            [ a0  : fix,
              i0  : xptr(p, 2*n),
              dsp : r,
              csp : s
            ] :: s
    ]
    a0 = 0.0; i0+=1
    do (i7) {
       { k : int  | 0 <= k < n}
       [ a0  : fix,
         i0  : xptr(p, 2*k),
```

```
20            dsp : int(k) :: r
21          ]
22          x0 = xmem[i0]
23          a0 += x0; i0+=2 ; xmem[i0] = x0
24       }
25       ret
```

A.5 Sum Over Complex Numbers

The code sums both the real parts and the imaginary parts of an array of complex numbers. The code uses the prefetch idiom and makes two out-of-bounds reads. The type annotations are drawn from the baseline type system.

Types have been elided in this version, because the implementation automatically insert the types that should be repeated.

```
1    sum_re_im_prefetch:
2        (s, r)
3        { n : int |  n > 0}
4        [ i0  : fix xarray(2*n),
5          i7  : int(n),
6          dsp : r,
7          csp : [ a0  : fix, b0 : fix,
8                  i0  : fix xarray(-2),
9                  dsp : r,
10                 csp : s
11               ] :: s
12       ]
13       a0 = 0.0; b0 = 0.0
14       x0 = xmem[i0]; i0+=1
15       x1 = xmem[i0]; i0+=1
16       do (i7) {
17          { k : int  | 0 <= k < n}
18          [ a0  : fix, b0 : fix,
19            x0  : fix, x1 : fix,
20            i0  : fix xarray (2*(n - k - 1)),
21            dsp : int(k) :: r
22          ]
23          a0 += x0; b0 += x1; x0 = xmem[i0]; i0+=1
24          x1 = xmem[i0]; i0+=1
25       }
26       ret
```

Bibliography

[1] Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 74–85, January 2003.

[2] Analog Devices. *Digital Signal Processing Applications Using the ADSP-2100 Family*. Prentice Hall, 1990.

[3] David Aspinall and Adriana B. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 2003. Special Issue on Proof-Carrying Code. To Appear.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[5] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[7] William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of a systems with non-linear constraints. In *Computer Aided Verification*, pages 316–327, 1997.

[8] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *PLDI 2003: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–219, June 2003.

[9] James Cheney and Greg Morrisett. A linearly typed assembly language. Unpublished draft, available from `http://www.cs.cornell.edu/people/jcheney`, 2003.

[10] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Deriving preconditions for array bound check elimination. In Olivier Danvy and Andrzej Filinski, editors, *Proceedings of Second Symposium on Programs as Data*

Objects (PADO-II), volume 2053 of *Lecture Notes in Computer Science*, pages 2–24. Springer-Verlag, 2001.

[11] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965. ISSN 0025-5718.

[12] Karl Crary. Toward a foundational typed assembly language. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, January 2003. ACM Press.

[13] Karl Crary and Stephanie Weirich. Resource bound certification. In *Symposium on Principles of Programming Languages*, pages 184–198, 2000.

[14] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[15] DSP-C, 1998. *DSP-C*, release 9.9 edition, October 1998. An extension to ISO/IEC 9899:1990.

[16] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, San Antonio, Texas, January 1999. ACM Press.

[17] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[18] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter System (On formal undecidable theorems in Principia Mathematica and related systems). In *Monatshefte für Mathematik und Physik*, volume 38, pages 173–198, 1931.

[19] Henrik Grauslund. Linear decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, May 2000. Reference IT-E 843.

[20] Dan Grossman. Existential types for imperative languages. In *Eleventh European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 21–35, Grenoble, France, April 2002. Springer-Verlag.

[21] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. *Lecture Notes in Computer Science*, 2071:117–145, 2001. ISSN 0302-9743.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation, Berlin, Germany*, June 2002. Extended version in Cornell CS Technical Report TR2001-1856.

[23] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[24] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[25] John Edward Hopcroft and Jeffrey David Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[26] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[27] ISO/IEC 9899. *Programming languages – C*, 1990. ISO/IEC 9899:1990.

[28] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, Monterey, CA*, June 2002.

[29] P. J. Landin. The mechanical evaluation of expressions. *Computer journal*, (6):308–320, 1964.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996.

[31] Ian V. McLoughlin. DSP software development. *Linux Journal*, (61), May 1999. Available from `http://linuxjournal.com/article.php?sid=3266`.

[32] Jesper Blak Møller. *Symbolic Model Checking of Real-Time Systems using Difference Decision Diagrams*. PhD thesis, IT University of Copenhagen, April 2002.

[33] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, Daivd Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999. INRIA Research Report 0228.

[34] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[35] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, USA, January 1998. ACM Press.

[36] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[37] Michael Norrish. Kananaskis release of the theorem prover HOL4. Homepage: `http://www.hol.sf.net`.

[38] Michael Norrish. Personal comunication, 2003.

[39] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–224, 1999.

[40] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[41] Lars Pareto. *Types for Crash Prevention*. Phd. dissertation, Chalmers, Göteborg University, 2000.

[42] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[43] Mojzesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zählen, in welchem die addition als einzige operation hervortritt. *Comptes-Rendus du I Congres de Mathematiciens des pays Slaves*, pages 92–101, 1929.

[44] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[45] William Pugh and David Wonnacott. Experiences with constraint-based array dependence analysis. In *Principles and Practice of Constraint Programming*, pages 312–325, 1994.

[46] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, San Antonio, Texas, January 1999. ACM Press.

[47] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[48] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark*, July 2002.

[49] S. A. Seshia and R. E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In Jr. W. A. Hunt and F. Somenzi, editors, *Computer-Aided Verification CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 154–166. Springer-Verlag, July 2003. Available as `http://www.cs.cmu.edu/~bryant/pubdir/cav03c.ps`.

[50] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 366–381, March 2000.

[51] Kedar N. Swadi and Andrew W. Appel. Typed machine language and its semantics. Preliminary version, July 2001.

[52] David Walker and Greg Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Workshop on Types in Compilation*, number 2071 in Lecture Notes in Computer Science. Springer-Verlag, March 2001.

[53] David Patrick Walker. *Typed Memory Management*. PhD thesis, Cornell University, January 2001.

[54] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[55] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Computer Science Department, Oregon Graduate Institute, July 1999.

[56] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, September 2001. See also [55].