Linköping University | Department of Electrical Engineering
Master's thesis, 30 ECTS | Datateknik
2023 | LIU-ISY/LITH-ISY-EX-A--23/5624--SE

Wordlength inference in the Spade HDL

– Seven implementations of wordlength inference and one implementation that actually works

---

Ordlängdsinferans i Spade HDL
– Sju olika implementationer av ordlängdsinferens och en implementation som faktiskt fungerar

Edvard Thörnros

Supervisor : Frans Skarman
Examiner : Oscar Gustafsson

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Compilers, complex programs with the potential to greatly facilitate software and hardware design. This thesis focuses on enhancing the Spade hardware description language, known for its user-friendly approach to hardware design. In the realm of hardware development data size – for numerical values data size is known as "wordlength" – plays a critical role for reducing the hardware resources. This study presents an innovative approach that seamlessly integrates wordlength inference directly into the Spade language, enabling the over-estimation of numeric data sizes solely from the program's source code.

The methodology involves iterative development, incorporating various smaller implementations and evaluations, reminiscent of an agile approach. To assess the efficacy of the wordlength inference, multiple place and route operations are performed on identical Spade code using various versions of `nextpnr`. Surprisingly, no discernible impact on hardware resource utilization emerges from the modifications introduced in this thesis.

Nonetheless, the true significance of this endeavor lies in its potential to unlock more advanced language features within the Spade compiler. It is important to note that while the wordlength inference proposed in this thesis shows promise, it necessitates further integration efforts to realize its full potential.

Acknowledgements

No work is never truly ones one, there are always others that inspire and teach you new things. I want to specially thank my supervisor Frans Skarman for being very supportive and understanding. Alicia Peterson deserves a special mention for being very huggable when there was a lot of thesis to write. I also want to thank the grumpy old rainbow-man who doesn't understand recursion. Finally Chocolate for always being there for me.

Contents

Chapter 1

Introduction

Computers are the working horse of the digital age – the usefulness of computers in data processing cannot be overstated. A computer is of course a very general term and referred to an occupation in the times before digital computers [1]. Being able to do computations fast is important. One way to do computations very fast is to make dedicated hardware for them. Dedicated hardware can run fast, requires little power but is expensive to produce single circuits. There is however an alternative, FPGAs (Field Programmable Gate Arrays). An FPGA is a piece of dedicated hardware that can be programmed and is today used a lot in the military and for hardware prototyping. FPGAs are programmed very differently from "traditional computers" and there are a multitude of programming languages for this domain – these languages are called HDLs (Hardware Description Language). One of these HDLs is Spade – the topic of this thesis.

The Spade HDL focuses on usability and borrows much from modern programming language[**src:spadeAnHDL**, 2, 3]. By writing high-level code when describing hardware, we also open the door for optimizations and help from the compiler. This is the topic of this master thesis, a specific kind of optimization and user-help that can make Spade faster and safer than other alternatives like VHDL and SystemVerilog.

This thesis focuses specifically on wordlength inference using a novel approach of combining wordlength inference with typeinference. Wordlength is the number of bits to allocate to a value and when creating hardware you often have to specify this yourself. Getting this wordlength right everywhere can be tedious and time consuming or produce less efficient hardware. A small error in the wordlength may cause faults in the program and a too large wordlength wastes resources. Since the compiler has access to all the source code for the hardware, the compiler should be able to check the wordlengths everywhere and potentially optimize the wordlength where bits go unused. Putting these optimizations in the compiler allows code to be more general which aids reuse, a good practice in the modern software industry.

To tackle the problem of wordlength inference we need mathematical tools to approximate arbitrary mathematical functions. There are a lot of methods to pick from but the two simplest are interval arithmetic (IA)[4] and affine arithmetic (AA)[4] – both are used for numerical overestimation and gives a solid foundation for analyzing mathematical expressions.

1.1 Motivation

Creating more powerful tools allows us to do more powerful things with them. In the case of software this effect is even larger, anyone with a laptop and a dream can develop programs for anyone to use. One of these fundamental tools is the compiler and programming languages – no one in their right mind would use FORTRAN today when they have alternatives like Go, Rust or Python. Bringing this mindset to the hardware world can increase the productivity, usability and accessibility to custom circuits and accelerators. HDLs have the huge potential of improving all computation speeds in the world. It is debatable if this thesis alone will take us as far as to revolutionize the hardware industry, but it is certainly a step in the right direction for energy-efficient and faster computations.

1.2 Research Questions

The topic of wordlength inference is vague and large to further narrow it down three research questions have been determined. These questions aim to give the research focus and direction.

1. How can interval arithmetic and affine arithmetic be used to implement wordlength inference?
2. How does wordlength inference and optimization affect the number of LUTs?
3. Can wordlength inference be used to create more reusable code?

1.3 Aim

This thesis will implement wordlength inference in the Spade compiler using a combination of interval arithmetic and affine arithmetic. The implementation is then to be evaluated using a synthesis tool and compared to other Spade-programs without these optimizations.

1.4 Delimitations

The sample size of the programs is quite limited, there is no attempt made to generalize the findings to all hardware. This thesis is limited to Spade and FPGAs and will not consider optimizations on other kinds of hardware. Other more sophisticated error-estimation like ME-gPC [5] will not be studied.

Chapter 2

Background

This chapter gives a brief introduction to programming languages and programming language theory – though not nearly as comprehensive as a good book ([6, 7]) on the subject. Type checking and typeinference is also discussed. A birdseye view of computer-hardware and FPGA design is also given, for example what exactly is meant by wordlength.

2.1 Introduction to Compiler Structure

What constitutes a compiler is not always obvious. A compiler, in the most banal sense, takes an input program and outputs an output program. Some want the output to be "simpler" than the input, passing in a high-level program in C and outputting executable x86 machine code where x86 is considered "simpler" than C. The input to the compiler is often text, and we will assume this for the rest of this short introduction to compilers.

Each compiler is unique, but they often have a shared structure. The first step is often to do lexical analysis (also called lexing) in a lexer or tokenizer. Here characters are abstracted away, and the compiler has done the first processing of the text. When lexing the compiler often decide what piece of text is an integer-constant, a keyword, a string, etc. After the lexing the tokens are used to perform syntactic analysis – also known as parsing. During parsing the compiler understands structures in the program such as what is part of each function or correctly parsing the order of operations for mathematical expressions. The parsing usually produces an abstract syntax tree (AST). Though some compilers interweave these steps, they are usually there in spirit.

The Spade compiler has both a lexer step and a parser step which are located in different modules.

The compilers' work is not done yet. After all the syntactical analysis the semantic analysis can be started, semantic analysis is sometimes referred to as the "inner layers of the compiler" Here the compiler resolve identifiers, run type-checking and other static program analysis or do optimizations like moving around constants to avoid needless copies. The wordlength inference and optimizations will be an inner layer, the relevant details will be discussed in the Section 2.3.

After the compiler has finished optimizing, the output is generated and potentially lowered (made less complex) in multiple stages, often by translating to a simpler internal representation which in the end makes generating the final output of the compiler easier.

One "layer" is often called a "pass". Spade is a multi-pass compiler and perform these steps sequentially. [6, 7, 8]

Compilers have to construct a lot of complex information about the program. A visual some find helpful when reasoning about compilers is an imagined graph of "available information" Figure 2.1 tries to communicate the amount of information created in each step of compilation. The most important part being that we know a lot about the program in the type checking phase – more than when doing syntactical analysis.

Abstract Syntax Tree

General purpose programming languages need to be recursive in their structure – one may want a sub-expression inside another expression for example. Representing this structure inside a computer cannot simply be done with a simple list of values, the data-structures themselves must reflect the

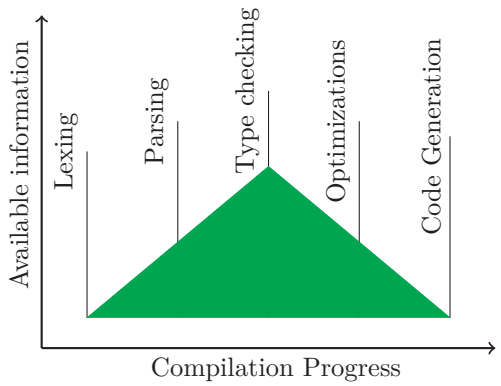

Figure 2.1: A visualization of the rough measurement of information present in each step of the compilation process. Type checking having the most information and lexing and code generation having the least amount of information.

structure of the language. One very popular way of storing this recursive syntactical information is by using an abstract syntax tree (often referred to as an AST). An AST describes the structure of the program.

```
<expr> ::= <num> | <expr> + <expr>

<num>  ::= 1 | 2 | 3
```

Figure 2.2: A simple grammar for integer expressions. Here an `<expr>` is either a `<num>`, so a simple number, or the it is a sum of two `<expr>`.

For example, consider the grammar for integer addition expressions described in grammar shown in Figure 2.2. The syntax understands whole numbers from 1 to 3 and the basic arithmetic operation addition (+). Most notable is that the syntax is recursive and expressions can contain expressions in themselves. This BNF-like syntax is a bit simplified and the grammar is not necessarily in a good format. A better grammar will handle order of operations and avoid ambiguities – which will be explained later.

The basic idea of the BNF is that we can expand each node – and all expansions are valid "programs" in the language. Here valid means syntactically correct – it does not mean the ideas expressed in the "program" are meaningful in any way. The rules expressed in the BNF say that an `<expr>` may be replaced with either `<num>` or `<expr> + <expr>` And that a `<num>` can be replaced with `1` or `2` or `3`. All valid programs are `<expr>` we can start expanding from there – the simplest program may be `1` since we can get this from expanding `<expr> -> <num> -> 1`. But we could also do the expansions:

```
<expr> -> <expr> + <expr>
       -> <expr> + <expr> + <expr>
       -> <expr> + <expr> + <num>
       -> <expr> + <expr> + 1
       -> <expr> + <num> + 1
       -> <expr> + 2 + 1
       -> <num> + 2 + 1
       -> 3 + 2 + 1
```

These expansions can get quite repetitive so special notation is often used to abbreviate. This syntax is also so simple that the format expressed here is needlessly powerful. But the important detail here is that we do not know if we expand the left-most `<expr>` or the right more `<expr>` when we apply the rule `<expr> ::= <expr> + <expr>`. To learn more we recommend any book on automatas or parsers ([6]).

We can write a simple "program" that is a part of the language described by Figure 2.2. Some small valid "programs" are: `1`, and `1 + 1`. But we will focus on the expression `1 + 2 + 3` – which has two additions.

Looking at the expression `1 + 2 + 3` it is obvious that the expressions should evaluate to 6. What is not obvious is how a computer should evaluate this expression. We can namely construct two different abstract syntax trees (and two different parse trees) for the program `1 + 2 + 3`. When parsing the program to a syntax tree we have to make the conscious decisions of what node is placed on top.

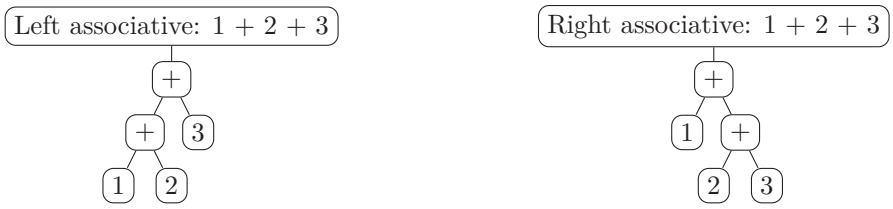

Figure 2.3: Two example syntax tree – illustrating ambiguity in the grammar and the difference between left and right associativity.

Figure 2.3 shows the two different associativities that causes `1 + 2 + 3` to be interpreted as either `(1 + (2 + 3))` or `((1 + 2) + 3)`. There is not much difference between these mathematical expressions – they are even considered equivalent since addition is associative for integers. The associativity property is unfortunately very rare in the real world of effectful-programming. If evaluating an expression in this language had side-effects – for example printing the result, different associativities cause different outputs. One of the programs print `1 2 3 3 6` while the other print `1 2 3 5 6` (ofcourse dependent on how the expressions are evaluated). This is what is called an ambiguity – and an abstract syntax tree makes it clear how the compiler internally understands this addition.

Making additions have side-effects is not the only way to get non-associative additions, one could use a less well-behaved addition like that found in the IEEE floating point standard – since any form of rounding is non-associative. Unfortunately infinite precision is quite rare when storage is finite.

There are different operations which can be done to an abstract syntax tree. In our very simple example of integer additions it makes sense to fold the constants and replace the entire AST with the constant 6 – but to do that we have to know the associativity of the `+`-operator.

Because of its flexibility the AST is a very popular format to output from the parsing stage. In the Spade-compiler the AST is annotated with type information as it is discovered – this annotated AST is then sent either to a language server or a code generation phase.

2.2 Syntax and Semantics

There are two concepts when talking about language, syntax and semantics. Syntax is the structure of the language, how the words are placed next to each other and what makes a valid sentence. Programming languages also have a notion of sentences and validity, and the word for syntax applies there as well. Syntax can also be used to describe natural languages.

Writing words down with valid syntax is all well and good, but they also need to be meaningful. The meaning of a sentence is called the semantics. This holds for programming languages as well. The expression `a * b` is syntactically valid, but lacks any kind of semantic without a definition for what '*' means and what `a` and `b` actually are, the example of `a * b` can be a multiplication between two terms but it can also be a declaration of a pointer variable in C.

Syntax and semantics are related, and the syntax can affect the semantics. The important distinction is that semantics is the meaning and syntax is the structure of the text.

2.3 Type Checking

Type checking is a way of making sure the program is internally consistent – there are no contradictions inside the program to the program itself. Type checking can be done in different ways with different pros, cons or preferences [9]. The type checker in Spade can infer types and deduce things about the program, like "the first argument is a 3-bit integer value, but you gave an enum" [**src:spadeAnHDL**].

For details regarding the Spade type checker, see Section **??** which discusses implementation details.

The type checker in Spade is a Damas-Hindley-Milner type checker [10]. This means it stops on the first error and can deduce types to their most general form. So if asked to type check the identity function (a function that takes one value and returns the value as is, the function does nothing) the type checker will be able to deduce that the argument can have any type, but that the type is the same as the type of the returned value, with only the body of the function.[10]

There is also a connected topic of typeinference – a program that guesses the types of expressions based on the context. A sufficiently good typeinferer can be used to check the types of the program and can easily be made into a type checker. It is in fact upon this idea that the Damas-Hindley-Milner typechecker works. In this work, we consider typechecking and typeinference the same operations – since Spade implements typechecking using typeinference. This equivalence is not the same for all typecheckers, since some only check types without inferring.

Unification

A Damas-Hindley-Milner type checker is a very simple typechecker and is based upon one simple idea: typeinference is a constraint satisfaction problem with *only* equivalence-constraints. The idea of the typechecking and typeinference is to simply find types that have to be equal for the program to function. Most of these small rules are painfully simple – like: "adding two numbers gives you a number" or "calling a function results in a type with the return value of the called function" What really drives these simple rules is a process called unification. Unification is a way to propagate these equality constraints. This is proven to be correct, given that all expressions without sub-expressions have a type.

This unification property is also found in logic-programming, which is not a coincidence but rather an effect of the CurryHoward isomorphism. [11]

We find examples useful to quickly get basic understanding of a topic, so we will outline the type checking of a simple program and how the unification step will work. We will typecheck the simple program in Figure 2.4.

```
1  fn add<A> : (A, A) -> A
2
3  fn main { add(add(1, 2), "abc") }
```

Figure 2.4: A simple dummy program to show unification. A simple "add" function is defined which takes two arguments of the same type and returns that type. Types are denoted with uppercase letters and language constructs are defined using lowercase letters.

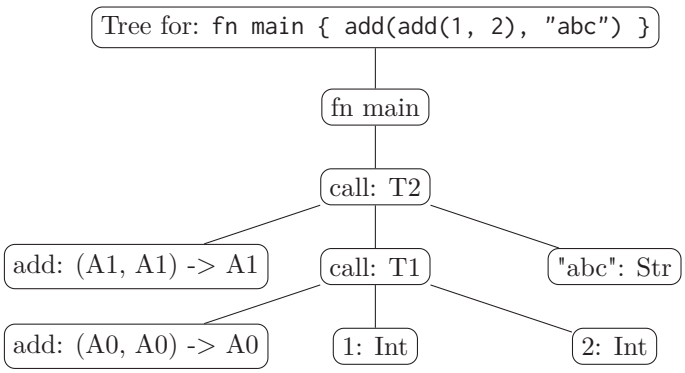

Figure 2.5: The syntax tree for `fn main { add(add(1, 2), "abc") }` with the generated type variables before we have annotated all the types.

Let us assume that we have two types in our language `Int` and `Str`. We start by giving each construct a unique type variable. (The unique type variable is mainly to aid explanation. There are more resource efficient ways of implementing Damas-Hindley-Milner.) We then evaluate the syntax tree bottom up – the syntax tree is shown in Figure 2.5. We will also assume the rules:

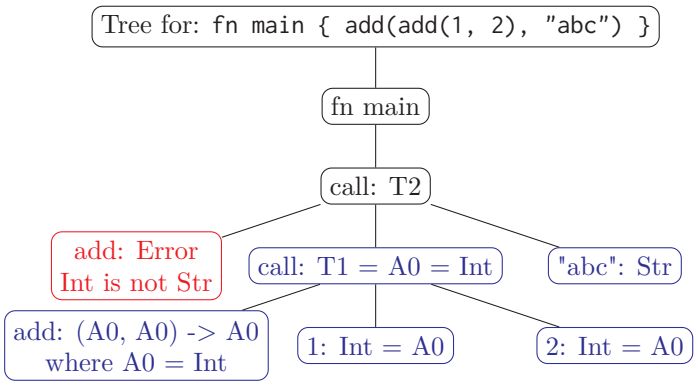

Figure 2.6: The syntax tree for `fn main { add(add(1, 2), "abc") }` with the generated type variables after we have inferred as far as we can. We see an error marked in red and finished nodes which are marked in blue. The error was found when we tried to unify `Str` with `Int` since the call requires both arguments to be the same type.

1. Integer-constants have the type `Int`.
2. String-constants have the type `Str`.
3. Calling a function means the types of the arguments can be unified with the parameters and the whole expression evaluates to the return type of the function. Evaluating rule (3) always gives a new type "A" we will keep track of these with numbered suffixes.

These three simple rules let us type check and typeinfer the expression in the main-function.

In this explanation the `:` operator denotes that an expression on the left has a known type on the right, so `1:Int` means the expressions `1` has type `Int`. We will also denote the unifications of types, so `A9=A8` means the type `A9` is the same types as `A8`. We first of all know the type of all the simple expressions – thanks to rules (1) and (2) – that means `1:Int`, `2:Int` and `"abc":Str`. We can now solve `add(1, 2)` by applying rule (3) to `add:(A0, A0) -> A0` and `add(1, 2):T1`, we can unify `(Int, Int)` and `(A0, A0)`, which gives `A0=Int`. We also get `T1=A0=Int` which gives us `add(1, 2):Int`. The more complex expression is now inferable `add(add(1, 2), "abc"):T2`, we unify `(T1, Str)` and `(A1, A1)` which gives us a contradiction, since `T1=Int=/=Str` – this will result in a type error which may be thrown by the compiler. This result is shown in Figure 2.6

In conclusion, unification lets us define two types as the same type and send all this information as far as we want in a program – there are programming languages with global type-inference for example. But the method described by Damas-Hindley-Milner is limited to equality-constraints which has some limitations in what types can be expressed.

Monomorphisation

After all the type checking and typeinference has been done monomorphisation can take place. This phase turns generic functions into concrete function – that is to say all functions with generics like `fn add<A>(x: A, y: A) -> A` are given concrete types like `fn add(x: i32, y: i32) -> i32`. This means the same code can lead to multiple function bodies. Monomorphisation creates a new instance of a function for each of the generic arguments given, and Rust does something similar. [12]

Monomorphisation is often considered a performance optimization, but for HDLs this step is strictly required since the hardware does not handle function pointers or any kind of indirect control flow. Hence, modeling virtual methods from object oriented programming is challenging. This means the resource usage will grow with the number of implementations for a virtual method, since we cannot know at compile-time which of these results is the right one (unless we use monomorphisation) – the scaling of this solution negates all the benefits of virtual methods and is not a good fit for FPGA development where resources are scarce.

The Lambda Cube

Typesystems are complex, and there are alot of them. But typesystems are also a very theoretical study, and comes from the idea of type theory. Type theory is a branch of mathematics that focuses on and reaching conclusions using types. A good place to start to understand at least parts of the field is to look at lambda calculus – a kind of "proramming language" similar to a Turing machine in that it is a mathematical construct for computation. Lambda calculus allows function definitions and from that natural numbers and boolean operations are constructed – this is quite a deep field and is not the exact topic of this work, so details will be left a bit hazy. But the lambda calculus is a very good playground for typesystems, since all computable computations can be reduced to an expression in lambda calculus and lambda calculus has functions and function calls as its only constructs.

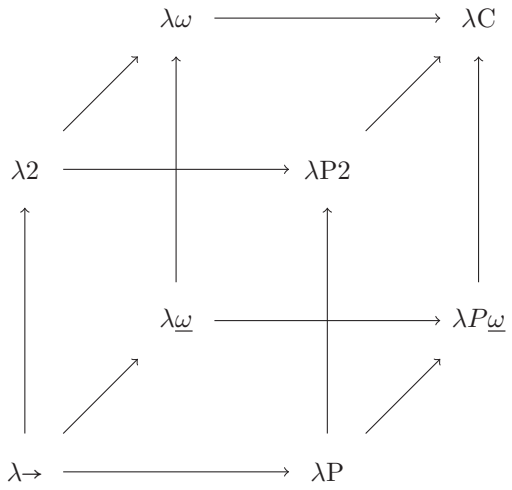

Figure 2.7: The lambda cube

When talking about typesystems for the lambda calculus you always limit the power of the language, but you can get some other niceties like bounded execution times. The lambda cube (as shown in Figure 2.7) is a way to categorize these different type systems in order of power. Each dimension of the cube is a different kind of feature for a typesystem. The bottom left of the lambda cube is the simply typed lambda calculus, where only terms working on terms is allows (this is usually called functions). The arrows going in the →-direction (not the same arrow as in λ→) mean there are types that can affect terms, known as dependent types and is considered by some quite esoteric but is mostly used in computer proof systems like Coq or Agda [13, 14] – these languages express mathematical truths not necessarily usable programs though the feature allows for more computation power. The arrows in the ↗-direction signals types that act on other types, this is mostly considered a convince and does not increase the power but can increase the expressiveness one for example higher-kinded types in Haskell or PureScript. Finally there are arrows in the ↑-direction which means terms that can affect types – or polymorphism as it is known in C++ or Java. In a sense, the lambda cube can be thought of as a map for typesystems and their complexities. The most powerful of these typesystems is the calculus of construction, and it has all these three typesystem features and it also has the most computational power of them all, meaning all of the other typesystems can be constructed by removing features for the calculus of construction.

Most typesystems for programming languages can be mapped to one of these theoretical type systems. Spade can be mapped to somewhere around $\lambda\omega$ since it has polymorphic functions and types that work on other types in the form of `ConstraintExpr` – though it is not as general as something like the Haskell typesystem so one could argue that Spade has partial higher kinded types, which puts it between $\lambda\omega$ and $\lambda 2$. Languages like Coq have all the bells and whistles of calculus of construction and is thus considered more complex. There is also things like Elm [15] which falls somewhere between $\lambda 2$ and $\lambda\omega$. This means Spade has a typesystem that is about as complex and powerful as that of Elm. So constructs that can be expressed in Elm should also be expressible in the Spade typesystem,

this is a good sanity check for language features that one wishes to add. Wordlength inference can be mapped to a record where each field is an integer and the field is present if the integer falls in the available range, the functions for these computations could not be expressed inside the Elm language since it is deliberately locked down, but if one had access the compiler one could unlock the full power and easily write these functions, this should give some theoretical basis for the possibility of this work.

2.4 Spade

Spade is an HDL that takes a lot of inspiration from Rust to create a more modern HDL (Hardware Description Language). Spade has syntax which mimics that of Rust and tries to remove problems people have when using other HDLs like System Verilog. One of the biggest features of Spade is the static type checking which allows programs to be verified before even being synthesized – creating a faster iteration loop. [**src:spadeAnHDL**, 2, 3]

Swim

The Spade-ecosystem has a build-tool call Swim. Swim orchestrates things like determine synthesis and place-and-route (PNR), running synthesis tools independently, installing dependencies, generating basic projects and compiling Spade-code. Swim is the best way to handle any Spade project.

Spade syntax

All programming languages have some kind of syntax – and Spade has taken a lot of inspiration from Rust in this domain but make additions and tweaks due to the different domain of the language. This section contains a short version of the relevant Spade syntax, there is more to the Spade-language but some short snippets are explained here.

Spade has multiple novel language constructs, there are: entities which can hold state while doing operations (think stateful functions to get an overview), pipelines that allow computations to stretch over multiple cycles and functions which are very much like functions in your favorite programming language. This guide will focus mostly on functions as a language construct since they are very well documented and most people who have written code has used something similar.

```
1 fn identity(x: int<5>) -> int<5> {
2   x
3 }
```

Figure 2.8: The definition of a function which does nothing with its argument.

The code in Figure 2.8 show a simple definition of a function which directly returns the argument passed to it. The function is defined by the `fn`-keyword similar to Rust. Just like Rust the type of the argument comes after its name, to the joy of the type theorists. The `int` type – which means integer and is the main focus of this work – takes a type argument that is passed inside of the angle-brackets (<>). The argument passed to the `int`-type is its wordlength, how many bits the integer fills up. A stark contrast to languages like C where you usually only have options that are powers of two. If the last statement in a function is an expression, it is automatically returned, again borrowed from Rust.

```
1 fn calculate(x: int<5>) -> int<7> {
2   let partial_result: int<3> = some_function(3);
3   x + 1 * 2 - partial_result
4 }
```

Figure 2.9: Some mathematical operations defined in Spade

Simple operations like multiplication, subtraction and addition exist and are expressed as one would expect as shown in Figure 2.9. Functions and entities can be called with parenthesis and any Rust-user should feel right at home. Statements are separated by semicolons.

```
fn generics<#A, #B, T>(x: int<A>, t: T) -> int<B> {
  if t == t {
    x + x
  } else {
    0
  }
}
```

Figure 2.10: An if-expression and some generic arguments passed to a function.

Spade has generics, which are bound to functions the function `generics` that is defined in Figure 2.10. There are two different kinds of generics, those that have a #-sign before them and those that do not. The # indicate a compile-time integer, and are used to aid the compiler with inferring the wordlength of expressions between functions. The remaining generics are type variables which are types passed to types. There are also if-expressions in the language that allow branching, these use the same implicit returns as the function bodies.

There are more language constructs in Spade, but these constructs are deemed irrelevant for this work – for a more through reference see the Spade-book[1].

The Spade Type checker

Since Spade is a statically typed language the compiler has a type checker. The Spade typechecker is a variant of Damas-Hindley-Milner and the intricacies of this class of type system is discussed in Section 2.3 which discusses typechecking more broadly while this section mentions the specifics of the implementation in the Spade compiler before any changes were made.

The Spade type checker walks through he syntax tree of the program applying rules to each node, unifying types (Section 2.3) which have to be equal for the program to be valid. Each expression in Spade has a corresponding type variable which holds the type of the expression. These type variables can also contain type variables inside them, so types with generics are encoded with their generic arguments. The implementation looks a bit like: `TypeVar(Id, Vec<TypeVar>)`, this is a bit simplified but the details are enough to understand the rest of this work. The unification of the outer type variable will also cause the inner type variables to be unified.

Besides type variables there are constraints and requirements – they do different things so the synonyms make them a bit confusing to talk about. `Requirement` is the least relevant, and handles methods and fields that are "required" to exist on types, but also that a certain type variable has to hold an integer of a certain value. After a `Requirement` is satisfied the affected type variables are updated and the `Requirement` is discarded.

There are also `ConstraintExpr`, which are used and referenced a lot more in this work. `ConstraintExpr` handle compile-time integers and have functionality for adding (`Sum`), subtracting (`Sub`), variables (`Var`), constants (`Integer`) and evaluating using $\text{floor}(\log_2 n+1)$ where n is the compile-time integer (`BitsToRepresent`) – the logarithm operation is used to find the number of bits a value needs to be represented. These constraints are solved in a similar way to the constraints, are removed when they are satisfied and relevant type variables are then updated. The `ConstraintExpr` lack syntax in the Spade language and cannot be expressed outside of the type checker.

Wordlength Inference in Spade

Most values in Spade take up bits or space in the run-time environment. Wordlength inference is mostly concerned with integers – so consider positive numbers without a fractional part. Consider a program with a counter that resets to 0 after counting to 3, we do not need 32 bits to represent it. The cost of storing a number with 32 bits compared to 2 bits could be large if it requires a larger FPGA, extra circuit components or a different power rating. Compared to software engineering of programs for general purpose computers where memory is considered abundant, FPGAs are limited in memory and computations use parts of the FPGA which causes HDLs to be designed differently to a general purpose programming languages.

[1] `https://docs.spade-lang.org/`

Wordlength inference is the compiler understanding what wordlength – the number of bits – is needed to store a value. Inferring this value well results in good resource usage and requires less manual intervention. Doing it poorly or not at all either requires users to manually specify the wordlength of each value, or produces hardware that is inefficient.

In Spade integer types are written using the `int` keyword, and wordlength is specified by passing a generic to the integer type. `int<3>` specifies an integer that spans 3 bits and holds values between $-(2^2) = -4$ and $2^2 - 1 = 3$. Spade also infers wordlengths based on the number of additons and multiplications – so one addition adds one extra bit of required precision while a multiplication adds the wordlengths together. This approach means that equivalent expressions require different wordlengths depending on the number of additions and multiplications used to express it.

```
1 fn f(a: int<3>) -> int<4> {
2   a + 1 + 1
3 }
4
5 entity main(clk: clock, rst: bool) -> int<4> {
6   f(0b111)
7 }
```

Figure 2.11: A simple Spade program that does not compile, showing the current limitations of wordlength inference.

```
error: Type error
  |- src/simple\_fault.spade:1:27
  |
1 |   fn f(a: int<3>) -> int<4> {
  |                      ------ int<4> type specified here
  | |-------------------------^
2 | |   a + 1 + 1
3 | | }
  | |-^ Found type int<5>
  |
  = Expected: 4
  =       in: int<4>
  =      Got: 5
  =       in: int<5>

Error: aborting due to previous error
```

Figure 2.12: The output from the compiler when trying to compile the program in Figure 2.11

Consider the program in Figure 2.11. The function f adds three values together, two of which are known constants. We also know that $2^2 + 2 < 2^3$ – we should be able to fit the result of the addition into the 4 bit word without loss of data. The compiler does not agree as seen in the compiler output in Figure 2.12 where it claims we need 5 bits to store this value. This problem may seem inconsequential since calculating constant expressions during compilation will fix this, as seen by the program in Figure 2.13 compiling without worries. Adding constants is only the tip of the iceberg that is this problem. The problem here is much deeper, and is a direct cause of the implementation of type checking and wordlength inference directly.

The Spade compiler implements a Damas-Hindley-Milner type checker – discussed in more detail in Section 2.3. Damas-Hindley-Milner runs in almost linear time (if implemented correctly) but is not complete. The completeness property for type checkers means there are programs which are correct but that the type checker will not recognize as correct. The programs from Figures 2.11 and 2.13 are an example of this. This observation is important since it means this is not a bug, but a limitation of the typechecking algorithm itself.

```
fn f(a: int<3>) -> int<4> {
  a + 2
}

entity main(clk: clock, rst: bool) -> int<4> {
  f(3)
}
```

Figure 2.13: "simple_correct.spade" A Spade program that does compile and is equivalent to the program in Figure 2.11

```
fn f(a: int<3>) -> int<4> {
  trunc(a + 1 + 1)
}

entity main(clk: clock, rst: bool) -> int<4> {
  f(3)
}
```

Figure 2.14: "trunc_correct.spade" A Spade program that does compile and is equivalent to the program in Figure 2.11, but here we used a truncation operation to remove bits from the integer.

There is also another way the Spade program can be changed to make it compile. Figure 2.14 adds a trunction operation to the expression in `f`. The truncation operation, called `trunc`, shortens the wordlength of an integer. This shortening sometimes causes information to be discarded – but not always. The code in Figure 2.14 shows a "safe" truncation, since no information will be discarded in this case. The `trunc`-operation can be used erroneously and cause faults which may be hard to find.

```
trunc<4, 3>(sb1010) =   sb010 = 2
 sext<4, 5>(sb1010) = sb11010 = -6
 zext<4, 5>(sb1010) = sb01010 = 10
```

Figure 2.15: 3 Short examples for operations in Spade which affect the wordlength. All the numbers are encoded using twos compliment without anyhidden bits, the wordlength is the number of 0s and 1s in the bit-string.

In addition to `trunc` there is also `sext` and `zext` which increase wordlengths. `sext` is a "sign extension" and increases the wordlength of the integer while keeping the sign byte the same. `zext` is a zero extension and pads the integer with zeroes to increase the wordlength. Some simple examples are available in Figure 2.15 which shows three examples on the binary integer `sb1010` – encoded here using two's-complement and we assume no hidden bits – which is -6 in decimal. Each of the operations take type-parameter, the Spade compiler usually infers these, the first type parameter denotes the ingoing wordlength and the second the outgoing wordlength. Of special interest is the sign extension, since it does not change the numerical value of this negative integer, the padding of the bit-string is done with the leftmost-bit (the "sign"-bit) to keep the numerical value. The "zero extension" naively inserts zeroes – which correctly keeps the values for positive numbers but here results in a wild 10 appearing. The truncation simply chops of the leftmost bits and calls it a day, if there is information there the information will be lost.

HIR-lowering

Figure 2.16 shows the overarching architecture of the Spade language. Since the level of abstraction between Spade and Spade's compile target is so different, translating between these abstraction levels is quite complex and is done in steps. The relevant step in this chain is called `HIR-lowering`, HIR

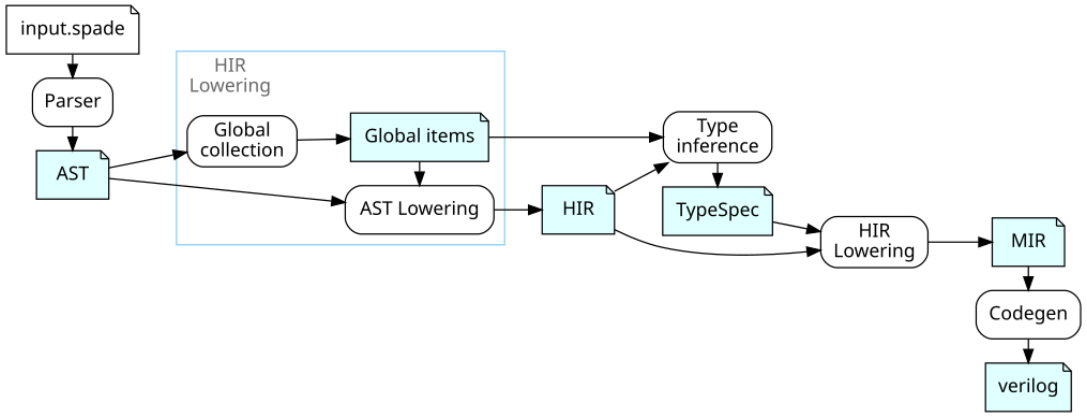

Figure 2.16: An over view of the Spade compiler, taken from the Spade architecture documentation[16].

stands for "Higher Intermediate Representation" and is the first of these steps, the later steps are not as relevant to the type checking but are very interesting when compiling. The HIR has been typechecked and monomorphised (Section 2.3)[16].

Rust

Rust is a statically typed general purpose systems programming language. The language has recently seen a lot of popularity and boasts features like high performance, reliable software and increased developer productivity. It is also the language that is used to write the Spade compiler. [17]

BigInt

Since Spade can express large wordlengths the maximum values cannot always be expressed in smaller sized integers. A solution to this is to use what is called "BigInt" which is an integer with arbitrary precision. This means that really large numbers can be represented. This representation is very slow compared to "normal" integers and the size of these BigInts can grow unbounded.

BigRational

Besides BigInt one can define a real number with arbitrary precision, this is trivial if you have a BigInt since all rational numbers can be expressed as a ratio between two integers. This representation is also very slow compared to "normal" floats and the size of these BigRationals can grow unbounded.

2.5 Interval Arithmetic, Affine Arithmetic and Self Validating Numerical Methods

Affine arithmetic (AA) and interval arithmetic (IA) are two common ways to estimate the value of arbitrary mathematical expressions. AA and IA can be applied to estimate bounds for mathematical functions and thus all things that can be modeled by mathematical functions. Since programs can be modeled using mathematical expressions IA and AA have a place in static analysis of programs, which is the focuses of this work. These methods are often referred to as over-estimation or self validating numerical methods. [4]

Interval Arithmetic

Interval arithmetic operates on intervals, as the name implies. A value – or in the context of a program, a variable – has a smallest and largest value it can assume. Consider `x = random_real()`, where `random_real` generates a random value in the range $[0, 1]$. We can express this in interval arithmetic as $\bar{x} = [0, 1]$, intervals will be denoted with a bar on top to separate them from the variables. Note

especially that the true value of x lies in the interval $\bar{x}$. In this example we know $0 \leq x \leq 1$, also written as $x \in [0, 1]$. These intervals can be added, negated, and so on, to give you an estimate of an arbitrary expression. The empty interval is also defined and written as $[]$. The empty interval usually denotes expressions or code that cannot be reached or evaluated – it a very sane default value for merging together multiple branches in a Spade match-statement, since if there are no matches we will in fact have unreachable code.

Interval Arithmetic: An Example and Limitations

Some of the rules for interval arithmetic are:

1. $n \times [a, b] = [n \times a, n \times b]$
2. $[a, b] - [c, d] = [a - d, b - c])$
3. $[a, b] + [c, d] = [a + c, b + d]$

We will be using the expression $2x + z - z$ as an example where $x = [0, 1], z = [1, 3]$. This means we have the expression after expanding the variables which we can calculate:

$$\begin{aligned}
& 2x + z - z \\
=^{\text{expansion}} \quad & 2 \times [0, 1] + [1, 3] - [1, 3] \\
=^{1} \quad & [2 \times 0, 2 \times 1] + [1, 3] - [1, 3] \\
=^{2} \quad & [0, 2] + [1 - 3, 3 - 1] \\
=^{\text{simplify}} \quad & [0, 2] + [-2, 2] \\
=^{3} \quad & [0 + -2, 2 + 4] \\
=^{\text{simplify}} \quad & [-2, 4]
\end{aligned}$$

We start by applying the scaling rule (1) – then the subtraction rule (2) and last the addition rule (3) giving us $2x + z - z = [-2, 4]$. This means the expression $2x + z - z$ with the context $x = [0, 1], z = [1, 3]$ always lies in the range $[-2, 4]$.

The conclusion we have reached is true, but the estimate is larger than it necessarily needs to be. Notice that subtracting the value z from itself should result in 0, which is a perfectly valid point and correct point. This is a limitation of the interval arithmetic. Interval arithmetic does not reason about the expressions that came before it and how they combine, and this limitation will exist if used to do static analysis of programs.

Affine Arithmetic

Affine arithmetic (AA) works similarly to interval arithmetic (IA), but has a memory of where values come from and can reason about their combinations at a higher level. But AA understand the relations between variables we evaluate if we provide the context. AA can more accurately calculate expressions like $a - a$ since it understands that a and a must hold the same value. That said, AA does not produce strictly better results than IA in all circumstances.[4]

Though affine arithmetic is more sophisticated it does not always produce better results, interval arithmetic can for some computations produce tighter bounds. There are other methods for overestimation that are considered more sophisticated like ME-gPC [5] and modified affine arithmetic, but the extra complexity can be added later if it is found to be needed. ME-gPC was not considered for this work due to its complexity – but it is a good extension to the techniques explained in this work. Affine arithmetic and interval arithmetic are the simplest methods that offer enough complexity to make this work interesting.

Affine Arithmetic: An Example and Limitations

In affine arithmetic there is a concept of noise symbols (e_i where i is a real number between $[-1, 1]$) and the numbers half width (x_i where i is a real number). A linear combination of these noise symbols is a reasonable way to represent a "number" when reasoning about affine arithmetic, $\hat{x} = x_0 + x_1 e_1 + x_2 e_2 + \ldots$. These terms can then be combined using similar rules to interval arithmetic. Of special interest is the first term, and the lack of a noise variable (e_0 is what one would have expected) – the first term can

be thought of as shifting the uncertainty up and down number line. Furthermore, we can consider the expression $\hat{x} - \hat{x}$, and since we have noise variables we get the expected result of 0.[4]

Here is an excerpt of relevant rules for affine arithmetic.

1. $[a, b] \Rightarrow x_0 = (a + b)/2, x_n = (a - b)/2$ where n is unique, this maps a range to its affine form $\hat{x}$
2. $n \times \hat{x} = \hat{z}$ where $z_n = n \times x_n$
3. $-\hat{x} = \hat{z}$ where $z_n = -x_n$
4. $\hat{x} + \hat{y} = \hat{z}$ where $z_n = x_n + y_n$
5. $\hat{x} \Rightarrow [a, b]$ where $a = x_0 + \sum_{1\ldots n} x_i e_i : e_n = -1$ and $b = x_0 + \sum_{1\ldots n} x_i e_i : e_n = 1$

Time for a simple example! Let us use $2x + z - z$ where $x = [0, 1], z = [1, 3]$. We start by expanding the context $x = [0, 1], z = [1, 3]$ from ranges to their affine form using (rule 1), $\hat{x} = 0.5 + 0.5e_x, \hat{z} = 2 + 1e_z$, we can then expand the expression and use our other rules.

$$
\begin{aligned}
& 2x + z - z \\
=^{\text{expansion}} \quad & 2 \times (0.5 + 0.5e_x) + (2 + e_z) - (2 + e_z) \\
=^{2} \quad & (1 + e_x) + (2 + e_z) - (2 + e_z) \\
=^{3} \quad & (1 + e_x) + (2 + e_z) + (-2 + -e_z) \\
=^{4} \quad & (3 + e_x + e_z) + (-2 + -e_z) \\
=^{4} \quad & (1 + e_x + 0e_z) \\
\Rightarrow^{5} \quad & [1 - 1 - 0, 1 + 1 + 0] = [0, 2]
\end{aligned}
$$

We start by applying the scaling rule (2), then we simplify subtraction to a negation and an addition using rule (3). Now we just have a final summation using rule (4) twice. After all this the expression can be changed to a range using rule (5) which gives us $[0, 2]$, which is exactly $2x$ – the answer that would have been preferred in Section 2.5. There is of course nothing magical going on here, all of these calculations are just simple algebra and we could just as easily have expanded the first expressions directly without using any special rules. However, this simplification cannot always be used.

When Interval Arithmetic Is Better Than Affine Arithmetic

Affine arithmetic may seem like the superior option in all cases – this is not true. Affine arithmetic has trouble with multiplication, and has to add extra noise in order to constantly overestimate. If we consider the simple case of multiplying two expressions in affine form $a + e_a$ and $b + e_b$ where e_a and e_b are noise variables we can use simple multiplication rules we get $(a + e_a)(b + e_b) = ab + ae_b + be_a + e_a e_b$, this is almost a valid affine form expression. But the pesky $e_a e_b$ term is illegal in AA. Since we have not defined what multiplying noise variables is (this is in fact the next improvement we can do and results in ME-gPC) though doing this will require handling squared noise variables which causes other problems which are out of scope. The way [4] define multiplication is slightly different – simplified they overestimate using $|(a \times b)| \leq |(|b| \times a + |a| \times b + \text{mid}(a) \times \text{mid}(b) + \text{rad}(a) \times \text{rad}(b))|$ where "mid" denotes the picking the value without a noise variable of an affine form and "rad" summing all terms which have a noise variable of the affine form. The extra terms are then added to a new noise variable which causes multiplications to become partially opaque – AA cannot understand the full result of a multiplication. This means any multiplication breaks $a - a = 0$ and adds noise. This noise causes us to increase the radius – and if the lower limit is something like 0 which is very special in multiplication we clearly get better results with interval arithmetic.

This extra noise is particularly relevant when handling large expressions. Interval arithmetic can fare a lot better here since multiplication does not add noise in the same way. One very concrete example is if all variables in an expression like $\prod x_i$ are in the range $[0, n)$ – affine arithmetic does not respect 0 in the same way and will give us a range poking out below 0 while interval arithmetic will correctly infer the lower bound of 0.

When concerned with estimating arithmetic expressions we can both get the pie and eat it. Since both interval arithmetic and affine arithmetic offer estimates which are guaranteed to hold all potential

values, we can take the subset of the guesses and still keep correctness. The method of taking the smallest subset is referred to in this report as AAIA – since we run both methods.

Opaqueness

When reasoning about expressions and languages with the use of programs one must simplify. One of these simplifications is to not expand everything to their most complex and detailed form and settle with an overestimation. One of these overestimations is referred to in this work as opaqueness. Opaqueness means we cannot see the inner workings of an expressions but are left with parts of the information about the values that can be held there. A source of opaqueness is reducing an AA-expression to a range, since we discard the relationship between variables and reduce it to just two numbers. Another source is the multiplication of AA-expressions where we introduce a new noise variable, but this opaqueness is only partial since we keep most of the information just some of it is diluted.

Yet another example is the values passed to a function, where we are reduced to reasoning about the larger types and not the specific values. For example consider a function `f` that takes a boolean – either `true` or `false` as an argument – and returns an integer with range `[0,50]`. Let us assume `f` is a pure function, thus we only have two possible return values. But the typesystem does not reason about the return type as a set of only two possible values, and to the typesystem the returned type is opaque since we cannot reason about the two possible values returned, we will need a set of possible values then like `{0, 50}` instead of a full range. This is another source of opaqueness that is perhaps more abstract but quite necessary.

2.6 Field Programmable Gate Array

A field programmable gate array (FPGA) is a kind of application-specific integrated circuit (ASIC) – an FPGA is reprogrammable which is not generally the case for ASICs. An FPGA is slower and draws more power than a “normal” circuit but FPGAs allow very fast iteration since they are programmable and are cheaper to produce in small volumes. This makes FPGAs ideal for prototyping hardware designs. An FPGA is made up of a grid of components all connected by configurable interconnections – each component can then be programmed and connected individually. [18]

An FPGA is very different from a general purpose computer or CPU which is what is mostly programmed by high-level languages. An FPGA is a circuit which means all steps are taken simultaneously – this allows a large amount of parallel computations and makes FPGAs ideal for things like realtime signal processing – which is why they see a lot of use in media processing and military applications [19]. These physical limitations also put restrictions on the HDL. This is why variables are immutable in Spade.

Synthesis

Synthesis is a step which tries to simplify the boolean functions that make up a logical circuit and in turn give out logical gates. This is done by analysing the expressions and solving a minimization problem. This is a step that tries to allocate as few gates as possible to build a circuit. In the case of FPGAs synthesis maps the HDL design to the primitive circuits available in the FPGA. Primarily using look-up-tables (LUTs) which store all possible inputs and outputs to a binary function. There are also dedicated blocks like multipliers and memories.

Place and Route

Place-and-route, sometimes referred to as PNR in this work is when the program is encoded as hardware and happens after the synthesis step, which requires routing the information in the actual chip and deciding what node is responsible for what computation. This is similar to how the routes are placed on a “normal” circuit and is considered a computationally hard problem. Since the problem of PNR is computationally hard, a stochastic algorithm is applied, meaning the same program may not generate identical encoded hardware. When trying to measure the resource usage of FPGA programs this step must be run multiple times, since one run can always be a fluke.

Chapter 3

Related Work

Wordlength inference is a well-studied topic. The focus is often on fractional bits like the method Doi et al. suggests in [20] where they try to remove bits from places where they have little value – this is separate from the goal of this work where correctness is of uttermost importance. There is also work [21] by Uguen which tries to apply C/C++ compiler optimizations for general purpose CPUs to HDL versions of C/C++ – the paper contains a lot of good discussions about FPGA hardware but the findings could do with a large sample size of programs. A similar work is done by Frigo, Gokhale, and Lavenier. Work on actually inferring the wordlength of expressions is more akin to typeinference which is well studied [10, 9, 23]. Though the logic system underlying the most practical parts of wordlength inference may be solvable by a very simple constraint satisfaction algorithm.

There is also some value to be found in the method of software development in this work which is inspired by the Agile development methodology [24]. Making the developer tooling better is something that also aids in producing better software faster – that is something we believe is non-controversial and that [24] agrees with.

This work stands firmly in the space between type checking, optimization, HDLs and methods for software development – a very weird space indeed.

Chapter 4

Method

The method chapter gives a high level overview of the method used in this work. More detailed steps are delegated to the results (Chapter 5) which contains implementation details and the evaluation of the implementation.

The problem is clearly defined from the previous chapters, especially Section 2.4 which goes into detail with examples. Alternatives for this implementation are then explored and presented to the project manager of the Spade compiler – which also happens to be the supervisor for this work – for evaluation and consideration. This allows a flexible approach to implementation where the goal is to find the best implementation. This way of working is a take on the agile methodologies.

After a viable solution is picked, it is time to implement that solution. Since the project is open source special care should be taken to make the changes easily accessible and available after this work is written. A list of the repositories and versions of each software are available in the Appendix A and the git-repository for this work[1]. Steps of the implementation and details relevant to the results are discussed in the implementation details sections in Chapter 5. A full evaluation for each of the potential implementations and details about these implementations that are deemed relevant and interesting are mentioned in Chapter 5. For full details it is probably best to read the source code.

The implementation is then evaluated using some simple expressions, a FIR-filter available at the thesis website[2] and spade-memory-display. The modified version of spade-memory-display is available in a forked repository[3]. The programs will also be evaluated by a place and route, this work uses nextpnr version `0.6-29-g54b20457` for all the synthesis and PNR done in this work. Relevant statistics from the synthesis and PNR step is then saved, and all the experiments are run at least 50 times to give some credibility to the results. The most interesting metric is the number of LUTs used in the final design since the LUTs would show where the designs could remove unnecessary complexity springing from the more complex input to functions – the number of LUTs is a very good proxy measure for the complexity of the circuit.

4.1 Finding Implementations and Creative Problem Solving

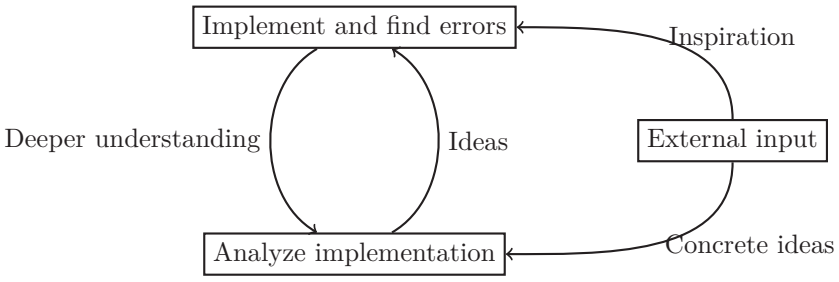

Figure 4.1: The creative process of solving complex problems according to the author.

[1] https://github.com/FredTheDino/thesis-spade-lang
[2] https://github.com/FredTheDino/thesis-spade-lang/tree/main/messing/fir
[3] https://github.com/FredTheDino/spade-memory-display-wli-fork

Solving complex problems is a creative process. Programmings languages are required to be complex since the ideas expressed in them are themselves complex. Complex problems require complex tools. The complexity requires that the programming language has a certain degree of quality – else the programming language quickly becomes useless and frustrating to use. To find a solution of sufficient quality requires trial and error and a good dose of analysis. Understanding partial solutions to a complex problem is a natural way of finding a solution that is higher in quality and hopefully good enough for the programming language to be usable. Figure 4.1 contains a visualization of how these steps integrate with each other.

Chapter 5

The Development Process

The problem of doing wordlength inference is a complex problem. This section contains a series of different implementations that failed in various ways, only the last implementation "worked" and is described in Section 5.8. The other implementations can be seen as the road to the final implementation – the value they bring is giving context and reasons for why the final implementation got its final shape.

Each section describes the thought process, implementation details of most important, new insights gained from this work and an evaluation of the final state of the implementation for each attempted solution. Explicit commit hashes are given to make it very clear what version is discussed.

5.1 Initial Requirements

Starting the implementation work we knew parts of what we wanted to achieve. We wanted a more sophisticated version of wordlength inference that handled mathematical expressions and required less manual intervention for wordlengths when writing Spade programs. The first step was understanding what was currently happening with the wordlength inference and where the problems where located. The goal of this implementation was to find out just how much we did not know about the problem of a more sophisticated wordlength inferrer.

1. Require fewer explicit `sext`, `zext` and `trunc` specified by the programmer
2. Make it easy to support unsigned integers – which is something the Spade compiler is hoping to add soon
3. Be as compatible with the current mainline Spade-compiler and easily switch between different versions of wordlength inference methods such as the old and any potential improvements using AA or IA
4. Allow for as much flexibility in language design as possible
5. Reduce resource usage of the resulting hardware
6. Integrate directly into the type checker

These requirements are prioritized so that requirement 1 is the most important and requirement 6 is the least important. The final implementation met all of the requirements except for 6 – since the wordlength inference was implemented in a module that operates on a type checked syntax tree.

Replacing the typeinference in the current compiler with something more powerful can potentially be the best solution. The current typeinference was regarded by some contributors as magic and was hard to work with and reason about – in some sense the module was questionable technical-debt. This poses a lot of language design questions, questions this thesis hoped to avoid since it may cause features to not be merged and made useful in the mainline compiler. Considering for example that a more powerful typeinference may not be desirable in the language, since with power comes complexity and more powerful type checking-systems like Haskell, PureScript or Coq. One need only take a walk on the lambda cube from Section 2.3. The more complex typesystems are known to make the problem of error messages harder – the antithesis of Spades goal of friendly error messages – but these more sophisticated typesystems offer great benefits to those who can wield them. To add more complexity and power to the typesystem is not a change to be made eagerly.

A separate wordlength inference module is not a completely obvious solution. Splitting the wordlength inference from the type checker has both pros and cons. Information from the wordlength inference has to end up in the inferred types from the typeinference – the modules need some kind of communication and communication causes overhead. However, the wordlength inference requires information about the expressions and variables used in them, which the typeinference currently throws away. Wordlength inference may also be helpful for the compile-time integers that exist in Spade – splitting the wordlength inference to a different module reduces the coherence. A few developmental excursions were made into potential easier solutions.

5.2 Taxonomy of the Solutions and a Brief Overview

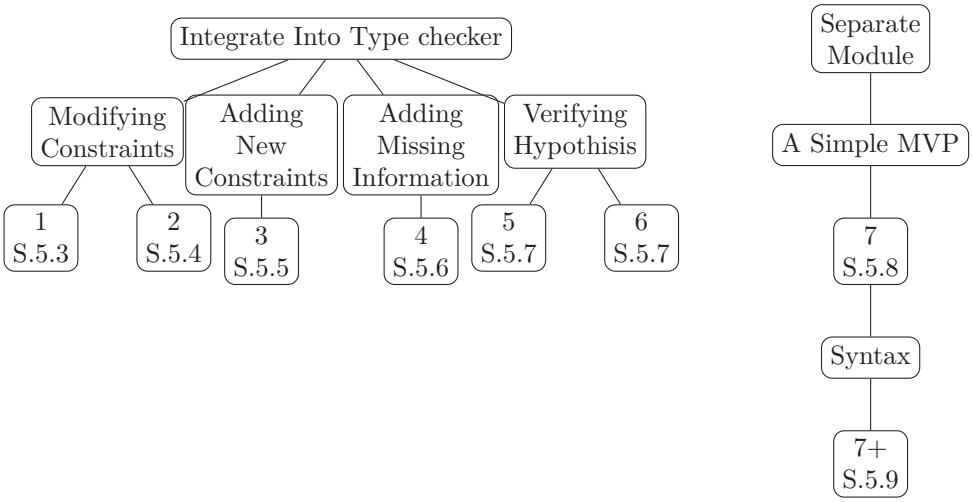

Figure 5.1: A overview of the different solutions and how their ideas relate to each other – the ideas are numbered and the relevant section for each approach is mentioned.

There were a lot of implementations that were tested and iterated upon as can be seen in Figure 5.1. The majority of attempts hoped to modify the existing type checker into doing the wordlength inference better – but these changes interfered with a lot of other language features like type level integer constants which are required for memories. After numerous attempts it was decided to try something new from a different perspective. A separate module for wordlength inference worked wonders and was simpler to implement, the changes are discussed further in Section 5.8.

Most of the implementations from Figure 5.1 were experiments and were not thought to be the solutions most likely to work. The more encompassing and maybe easier solutions have always been adding a separate module for wordlength inference – in a similar way to how the linear types are checked in Spade at the moment – or adding a whole new typeinference and type checker. The following sections list the different implementations and what was learned from them.

5.3 The First Implementation – Naive

It was not immediately obvious how affine arithmetic would fit into the Spade typesystem. What did seem like a good idea at the time was adding ranges to the integers in the typesystem. Hence, the initial naive implementation used the `ConstraintExpr` that existed for compile-time integers and tried to force it into doing better wordlength inference. `ConstraintExpr` was the basis for the wordlength inference used in the mainline Spade compiler which simply added one to each number and was therefore what immediately sprang to mind[1].

`ConstraintExpr`s were hijacked into supporting ranges, to see where the compiler started to complain and find other potential pot holes or errors. `ConstraintExpr` was changed to support addition,

[1]The changes for the first bash at the problem are placed on the git-branch `the-simplest-implementation` and has git-hash `74c966a0317aa738017d2edf15def4719fe8dc95`.

negation, and multiplication operate on ranges with a a high and a low value – in contrast to how `ConstraintExpr` worked with a single value prior. The type checking of binary expressions was also changed to add these new kinds of constraints and checks.

This change broke a lot of things in the compiler in interesting ways. Array lengths no longer worked since they also used these kinds of `ConstraintExpr`. The types of some expressions also failed to infer which caused the `HIR-lowering` to fail. The reason for the missing types was because there are choices to be made when type checking some expressions, what wordlength should actually be used when there are multiple candidates. There was also an attempt at mitigating the missing types by introducing an extra solver stage for the `ConstraintExprs` – this stage ran after a function was typechecked and tried to pick the smallest size of all constraints and did not work.

The change was also too large to allow as much compatibility as possible with the mainline compiler, broke language features and added more complexity to the constraint system. These changes signaled that changing the `ConstraintExpr` to operate on ranges would be too much to ask if one wanted any kind of backwards compatibility. The problem of failing to infer the wordlengths for some integers was also noted here, and that it usually was the symptom of not giving the typesystem enough information to deduce all wordlengths – the separate solver did not help it either.

How affine arithmetic can be added into this kind of code was still to be decided, since the type checker is a logical system it has to run backwards which is something affine arithmetic cannot do.

The implementation outlined here did not work – it was also never expected to work. It gave very clear hints as to where to dig deeper, where the understanding was sub par and some of the hidden requirements for adding wordlength inference.

5.4 The Second Implementation – Another Naive Stab

Full with unmotivated optimism the `ConstraintExpr` was extended to support a `BitsToRange`-value – which worked as the opposite of the `BitsToRepresent` expression. `BitsToRange(X)` was a constraint that turned a wordlength into the maximum positive value representable for a signed integer with `X` bits. The idea was to allow moving between wordlength and a max-value in the type-level expressions. This would hopefully leave the array code working, while allowing the expression code to infer wordlengths since the added constraints would add information. An attempt at fixing the problems outlined in the first solution. This idea and implementation was not supposed to solve the entire problem completely. The second attempt was not a fit solution for the entire problem since it only solves the sub-problem of very naive interval based wordlength inference – affine arithmetic was nowhere to be seen as mentioned. The ambition was to solve the easiest parts of the problem first since if the idea cannot handle the easy parts it is not worth thinking about the hard parts[2].

```
1  def CheckAddition(Op=Addition, Lhs, Rhs):
2    Constraint(Result = ToRepresent(MaxRange(Lhs) + MaxRange(Rhs))
3    Constraint(Lhs = ToRepresent(MaxRange(Result) - MaxRange(Rhs))
4    Constraint(Rhs = ToRepresent(MaxRange(Result) - MaxRange(Lhs))
5
6  def CheckIf(Condition, True, False):
7    Unify(True, False)
8
```

Figure 5.2: Pseudo code for the wordlengths of the 3 unknowns in an addition (the result, the left hand size (LHS) and the right hand side (RHS). Another function is sketched for checking of If-Expressions where `True` and `False` are the types of the results of the respective branches. `ToRepresent` is the same as `BitsToRepresent` and `MaxRange` is the same as `BitsToRange` – the names were changed to make it easier to read at a glance. `Constraint` signals a new constraint that should be added.

Figure 5.2 contains pseudo code for the changes introduced in this second idea. It adds 3 constraints for an addition and 1 unification for an If-Expression. Now let us ponder the case where we have the

[2]Changes are available on the git-branch `the-second-simplest-thing` and has git-hash `3d92c0e4b28b64104f700c3d299f62e7938cb016`.

following if-expression `if ... { a } else { a + 4 }` in a context where `a` is an integer of wordlength 1 (`a: int<1>`). We unify both If-branches, which causes `a` to be part of the result and the `Lhs`. We can infer `Lhs=1` and `Result=1`, which changes the first constraints to `1 = ToRepresent(MaxRange(1) + 4)`. This constraint gives a contradiction which shows how inflexible this solution is. Ideally this piece of code would compile fine – since we can easily fit $a + 4$ into the same wordlength as a. But to get this to type check we require a `sext`, which highly contest out first requirement.

The constraints listed in 5.2 are all equality-constraints (`=`) and for this idea to work less-than-or-equal constrains (`<=`) would be more suitable. Less-than-or-equal constraints may be required for wordlength inference, but these cannot be expressed in the Spade typsystem as is. The problem of adding affine arithmetic is still unsolved. We want to encode the operations and additions into a structure to support affine arithmetic – which would then require a new system to work in parallel to . We also need to carry the information for the intervals of variables somewhere else, since the current typesystem really disagrees with encoding ranges into int-types – a new construction may be required.

5.5 The Third Implementation – Considering Equations

The third attempt focused on adding equations and added a separate constraint language. These constraints were part of the typeinference[3].

A new set of constraints was added giving `ConstraintExpr` and `Requirement` a new friend called `SizeExpression`. Aside from the obvious technical-debt with three different kinds of constraints that express similar ideas – the solution was deemed too "ugly". The constraints needed for `SizeExpression` were required to be feed forward by the typeinference – following a completely different approach to the other requirements which were added to lists. This caused a large amount of changes in the typeinference which was preferably avoided.

We also started to suspect that the wordlength inference may *need* to be a separate module, split up from the typeinference and run after all the types have been inferred. This idea did however not live long, and might even be a mere curiosity.

5.6 The Fourth Implementation – A Desperate Attempt

Almost out of desperation for getting something to work. Just to make sure this whole ordeal was not impossible – these changes were made. These changes are also the most insightful in all of these experiments[4].

The pseudo code in Figure 5.3 describe how an addition expression is wordlength inferred according to the fourth method. The code sent extra information – besides of the wordlength stored in the type of each integer. This extra information also resulted in extra type variables which denoted the maximum value – these type variables can not be explicitly manipulated and are hidden from the user of the language. These type variables were not strictly bound to anything and were hard to reference since they can not be found from the type they were constrained to if the constraints were satisfied and thus discarded. There was not a problem with the kind or number of constraints since `LhsMax` and `RhsMax` was always known or bound directly to a size.

This was the first approach to produce better wordlength inference than the mainline-Spade wordlength inference and handled expressions such as `a + 1 + 1` as one would expect by not increasing the wordlength by 2 but by 1 when the range of `a` was larger than `[0, 2]`. There is also a very concrete way to introduce negative ranges – just add another type variable. This means the solution clearly solves requirements 1, 2, 3, 4 and 6 from Section 5.1 – the resource usage was never investigated.

This solution also introduced problems with the hidden type variables. Since the type variables for the maximum size cannot be referenced in any meaningful way from the type of the expression it was deemed be difficult to explain to the user of the language why a type-conclusion was reached. But since constraints were discarded after they were satisfied the relations between the type variables

[3]Changes are available on the branch `the-third-simplest-with-equations` with the git-hash `83d1da48f5010767b9a96aea0a5bca13b7415084`.

[4]The changes are available on the git-branch `the-fourth-attempt-now-with-equivelence` and has git-hash `ba0fa1baab56a725c31f703204da3b7d5f44380a`.

```
CheckAddition():
   # Take our best guess of a max value
   LhsMax = PassedLhsMax or Largest(LhsSize)
   RhsMax = PassedRhsMax or Largest(RhsSize)

  # We know the maximum values add to give a new maximum
  Constraint(ResultMax = LhsMax + RhsMax)
  # The maximums implies a size for each of the values
  Constraint(ResultSize = ToRepresent(ResultMax))
  Constraint(LhsSize = ToRepresent(LhsMax))
  Constraint(RhsSize = ToRepresent(RhsMax))
  # Return our best guess to make the expressions around aware of us
  return ResultMax
```

Figure 5.3: Pseudo code that described the idea for inferring wordlengths with extra information – what was used in the fourth attempt. Here `PassedLhsMax` and `PassedRhsMax` are type variables that sometimes exist and are results of type checking similar expressions – one can envision some code that calls `CheckAddition` for each of the subexpressions in the binary expression we check addition for. `ToRepresent` convert a maximum value to the smallest wordlength that can fit it. `Largest` is almost the opposite of `ToRepresent` and change a value to the maximum value such a wordlength can hold

and the maximum size was also discarded – this meant that range information can not be recovered. The problem of integrating AA was not prodded in since this solution was not good enough – due to the potentially unsatisfactory error messages. The problem with equality constraints (`=`) from Section 5.6 is also relevant, since it may force a value to have multiple wordlengths – a neater solution is still less-than-or-equal constraint (`<=`).

5.7 Other Implementations – Just Curiosities

There were two more attempts at an implementation inside the type checker. Both these experiments yielded little of interest since they mostly verified the previous claims, mostly the implementations from Sections 5.3 and 5.4. They were also very shallow changes since the more promising change outlined in Section 5.8 was started on instead. These ideas mainly focsed on re-testing old ideas and are of little interest for the final results[5].

5.8 The Seventh Implementations – a Separate Module

After doing a lot of thinking and reasoning we decided to see how a separate module would look – in theory, this gives maximum flexibility in how the wordlength inference worked. Using a separate module for the wordlength logic also meant changing it in the future is easier[6]. Since the type checker would not be changed as much old programs can still be compiled and thus compared with and without the more advanced wordlength inference.

As a basis for the wordlength inference there is a simple type called `Range`. `Range` can be directly mapped to the interval from IA. The datatype is implemented using `BigInt` to make sure no overflow occurs. There is also a type called `AAForm` which maps directly to the form of AA expressions – `AAForm` is implemented as variables and constant scalars which are `BigRationals`. The decision of using `BigRational` was to avoid even more noise in the AA implementation.

The idea of the implementation was to walk the entire AST a second time after type checking had run and monomorphisation was completed – each entity was given to the wordlength-inference

[5] Available on the git-branches `the-fifth-attempt-now-without-returns` and `the-sixth-attempt`

[6] Changes are available on the branch `the-seventh-attempt-almost-as-simple-as-attempt-one` with git-hash `889afd61a59f04f60730964b8ae7a2703110dd99`, these changes were merged and is available in the PR hosted on `https://gitlab.com/spade-lang/spade/-/merge_requests/200`.

```
def Visit(Construct):
  Constraint = match Construct:
    # Variables are already resolved so we just need to make sure the inference
    # has id's that match 1:1 with the type checker.
    case Var(VarId):
      Var(VarId)

    # Constants are simply the constant, the range of these are known here.
    case Constant(Constant):
      RangeFromConstant(Constant)

    # Constants are simply the constant, the range of these are known here.
    case Assignment(Var, Expr):
      Constraints[VarId] = Visit(Expr)
      NoInformation

    # Binary expressions need to be recorded, we want to build up a complete
    # list of the constraints for each type.
    case BinaryExpr(Op='+', Lhs, Rhs):
      Addition(Visit(Lhs), Visit(Rhs))
    case BinaryExpr(Op='*', Lhs, Rhs):
      Multiplication(Visit(Lhs), Visit(Rhs))

    # Bitwise and is very hard to reason about and we make a very pessimistic
    # guess
    case BinaryExpr(Op='&', Lhs, Rhs):
      Union(BitManip(Visit(Lhs)), BitManip(Visit(Rhs)))

    # Uniary expressions also need to be encoded.
    case UnaryExpr(Op='-', Operand):
      Negation(Visit(Operand))

    # Flow control constructors need to be handled differently, for an If we
    # combine them and realize that the result must lie inside the union of both
    # branches - we return either True or False but we cannot know which for
    # all cases.
    case If(Condition, True, False):
      Visit(Condition)
      Union(Visit(True), Visit(False)

    # Note that functions became opaque, which is a large limitation
    case Call(Callee, Args):
      for Arg in Args: Visit(Args)
      # Visit(Callee) - we would love to return some information here, but it has been discarded earl
      NoInformation

  # Register this constraint, since *all* expressions have types we want as
  # many constraints as possible.
  Constraints[Construct] = Constraint
```

Figure 5.4: Pseudo code for visiting the entire syntax tree – some details have been omitted. The algorithm handles each node separately and visits all the inner nodes, taking care to visit *every* expression in the syntax tree and accumulating constraints for the types they relate to.

module[7] – this built up constraints described in the algorithm in Figure 5.4. The algorithm makes sure to visit *every* expression in the AST and record down their relation to each other and since we have already typechecked we know which expressions need to constraints built up. The algorithm in Figure 5.4 produces a mapping between language constructs and constraint equations relating it to expressions that have a relation. This mapping is almost what we want, and we can use it to produce the mapping from a type variable and an equation which describes the expressions size as related to other expressions. We can, of course, have multiple equations for the same type variable as long as they are consistent with each other. After all of these constraint equations have been gathered up we can pass the equations to a solver.

```
# Definitions of the operations that are constructed by the Visit function
def Var(A..B):
  A..B

def Addition(A..B, P..Q):
  (A + P)..(B + Q)

def Multiplication(A..B, P..Q):
  Min(A * Q, A * P, B * Q, B * P)..Max(A * Q, A * P, B * Q, B * P)

def Union(A..B, P..Q):
  Min(A, P)..Max(B, Q)

def Negation(A..B):
  -B..-A

def BitManip(A..B):
  # We assume the size of the integer doesn't change so discard all information
  # except the wordlength
  WordlengthToRange(RangeToWordlength(A, B))

def ToRange(A..B):
  A..B

def NoInformation:
  pass

```

Figure 5.5: An outline of the different range procedures, this explains how each constraint operation for the `Visit`-function should be evaluated if you want to use the IA method, each definition takes ranges and a low and a high range are broken apart in the arguments for readability, where the symbol before `..` denotes the lower bound and the symbol after the higher bound.

The IA method has is defined to do the following rudimentary operations from Figure 5.5 when we try to evaluate the constraint, if any variable is unknown we simply abort trying to evaluate the constraint and hope we get more information somewhere else. The `Addition` and `Negation` operations can be reversed trivially. The multiplication operation however, cannot be reversed – consider the example of `(1..2) * (-3..1) = (-6..2) = (2..2) * (-3..1)`.

The code in Figure 5.6 describe how the AA operations are implemented, some of them fall back to ranges which make them opaque and some of the operations introduce extra noise variables – these noise variables will cause inaccuracies in later results. Some of the operations are ill-defined and as a fallback the corresponding range operations are used – for example the `Union` and `BitManip`. The `Multiplication`-operation is also quite complex and are partially opaque. It is also clear that some operations cannot be undone since they discard information. The AA-form that is used for the AA

[7] The wordlength-inference module was called `spade-wordlength-inference`

method was implemented using BigRational, this means that arbitrary precision rational numbers can be stored.

The algorithm in Figure 5.7 outlines the solving of the wordlength constraints for the wordlength inference and has a run-time of $O(dn^2)$ where n is the number of equations and d is the largest depth of the equations – since we are bounded by running the loop a maximum of n times. The average run-time is significantly better since it can solve most sets of equations efficiently in fewer iterations. If the algorithm can make progress we also know that it will, since it evaluates all equations for each pass – this is a form of breadth first search since we constantly re-check the equations and disallow variables to shrink in size we always find more information with each iteration.

Evaluation of the body of the constraint equation using the functions `EvaluateUsingIa` and `EvaluateUsingAa` from Figure 5.7 return ranges, this discards information when using AA since the reduction from AA to range is lossy. The evaluation of the built up equation was implemented by simply replacing the variables with values when the variables were known, slowly withering away at the problem. This reduction to ranges caused variables to be opaque to the wordlength inference.

The inference algorithm from Figure 5.7 can be applied from the constraints gathered from the algorithm in Figure 5.4. This means we have visited the entire AST, picking out the expressions that are typed to be `int<_>` and construct algebraic constraints on the form `<var> = <expression>`. We can then feed these equations into a solver and use them to solve more equations – here we can pick between using using IA or AA. In the implementation this can be set when running the compiler[8]. We can then enrich the type checkers types with our new findings which let us reuse the rest of the compiler tool chain – very nifty!

Evaluating the Changes

The requirements from Section 5.1 were almost all satisfied by this solution – this solution requires fewer `sext`, is easier to troubleshoot, makes it simple to add in unsigned integers, allows swapping back to the old way of doing wordlength inference – the only exception being that it is not integrated into the type checker. That the wordlength inference is not integrated into the typechecker is both a pro and a con, it simplifies the implementation and makes it easier to change if one wants to make modifications to the wordlength inference semantics without affecting the typeinference as a whole. The wordlength inference module is required to *agree with* the typechecker and the code only hints about it in the unification step, instead of one slightly larger complex system we now have two somewhat smaller systems that may not disagree.

The implementation was also run on multiple small programs. Some were written to show which expressions can be wordlength inferred and some where written to measure the number of LUTs to see how the performance was different.

	OLD (Old version)	IA	AA
Average number of LUTs	179.6	176.9	177.1
Variance for number of LUTs	13.2	6.0	8.2
Largest number or LUTs	186.0	181.0	183.0
Smallest number of LUTs	171.0	171.0	167.0

Table 5.1: The number of LUTs after synteshsis and PNR for the project spade-memory-display with different versions of wordlength inference in table form. The experiment was run 51 times.

This implementation of the Spade compiler was then run on the example program `spade-memory-display`[9]. A simple bash script was made to record the number of LUTs used in the FPGA when the command was executed and this was stored in a file. The program was compiled 51 times for the three configurations: “Old Mode” – denoted by OLD, “Affine Arithmetic” – denoted by AA, “Interval Arithmetic” – denoted by IA. The number of LUTs are shown in Table 5.1.

[8] The different wordlenght inference options can be swapped using the environment variable `SPADE_INFER_METHOD`, the inference logic can also be disabled entirely, this was required since the Swim buildsystem is very peculiar about the flags passed to the compiler.

[9] using the swim command `SPADE_INFER_METHOD=XX swim pnr`

Table 5.1 shows a slight decrease in variance for the number of LUTs generated using IA (Interval Arithmetic) and AA (Affine Arithmetic). The average stays almost the same for all methods with OLD (The Old Version) being a bit higher than the others – but not by a wide margin.

These changes did show promise in the usability of the language. Snippets like the one shown in Figure 5.8 were type checked a lot better by the Spade compiler and in that sense a usability improvement can be seen, note especially the difference in the return type of the function `add_and_subtract` where the one typechecked with Affine Arithmetic always returns 0 and Interval Arithmetic returns 8 bits – compared to the Old approach which returns an integer with 8 bits but requires a manual sign extensions which may not be placed on another part of the expression. Looking at this code fewer truncations are needed since the compiler knows more about the actual size of the expressions. There is also an argument for the analysis part of this code: a function that returns an `int<0>` is probably not doing anything meaningful and should probably be optimized out.

This implementation does however leave a question unanswered: how does a user of the Spade language define these ranges? Since the changes discussed until now have not touched the syntax or the type checker (except disabling the old wordlength inference), the entire language as a whole is completely blind to the ranged-based wordlength inference. The current approach seemed to work well and it was decided that the best course of action was to extend this implementation with new syntax.

5.9 The Seventh Implementation Extended – Doubling Down on Ranges

After the success of the implementation in Section 5.8 more work was needed to communicate the intended ranges from a programmer. Thus began changes that updated the syntax and the type checker in Spade to support ranges. There was however a previous attempt at adding this kind of range syntax and are available in the Spade git-repository[10] that focused on the syntactical changes and avoided the typesystem changes, the merge request has since been abandoned. The new changes that needed to be made to the syntax took inspiration from the previous attempt. The new syntax and typesystem changes had to be scoped accordingly and a lot of the details and changes discussed are very opinionated and this thesis tries to not focus too closely on the design of programming languages and more on the actual implementation of the language features, though some background and reason will of course be given[11].

The previous attempt at changing the wordlength syntax tried to work from the syntax forward without interfering with the typesystem – this was deemed to be harder than expected. This new attempt tried to change both the syntax and the typesystem at once – this proved to be beneficial but the change was also quite large and caused a large break in the syntax and semantics of the Spade language. Basing the range based syntax on the newly implemented wordlength inference does alleviate some of the complexity, but it is also explicitly bound to this functionality. This also means that the old wordlength inference algorithm was broken and ceased to work, and no new results can be collected with it.

Another method besides AA and IA was introduced – this method is called AAIA and is the result of running both AA and IA in parallel and taking the subset range of both methods, this always produces the best results.

The syntax of the Spade language was first of all changed to parse `int` types differently, the syntax `int<L..H>` allowing ranges to be specified. The old syntax `int<W>` is still supported but was syntactic-sugar for `int<-`(2^{w-1})`..`$(2^{w-1})-1$`>` (the -1 part comes from all integers being signed in Spade) since this would give partial compatibility with older spade-program. In the previous work a lot more syntax was added but they were not needed to updating the typeinference. This change makes it possible to propagate the range information from one function to another in the form of a range – which is more precise than a wordlength which was what was available before. Figure 5.9 shows this new syntax in a small example spade-program.

The extra variables `a`, `b` and `c` are required to make the program compile, since the compiler has trouble type checking the variables between the expressions – so the expression `twice(twice(1))`

[10] `https://gitlab.com/spade-lang/spade/-/merge_requests/208`

[11] Changes that are available in the git-branch `wordlength-inference/push-the-changes-further` with git-hash `ee9980c6ec518dbdf0794dbb9193c5b1c9b6945e`

would not succeed to compile, this is a problem of the unification process and that there are insufficient requirements on some generics. The cause of this is the same as the main drawback of this implementation, the typesystem is not aware of the wordlength inferencce and cannot provide the information that is needed, these function calls become opaque to the wordlength inference since the return type of the function is not available. This problem is caused by a lack of information, which can be solved by either adding more information to the state of the typechecker so the wordlength inference can inspect more, or by integrating the wordlength inference into the typechecker directly (which was the method the old wordlength inference method), or by building a new typechecker inside the wordlength inference module. The first or second options are preferable over the third.

It is also worth noting that the extra types on lines 13 to 15 in Figure 5.9 can be larger according to the current rules – but must include the actually expected values. A line like `let z: int<-1..2> = twice(3)` would not compile, since the expected value of the function call is indeed 6 and 6 does not exist between -1 and 2.

The `int` type also had to change to work with two generic arguments – a lower and a higher bound. These types had to be inserted in the type checker. Making this change in full would require rewriting almost every single spade-compiler test, and would be a substantial amount of work and time, more time than this thesis has been allotted. The decisions to break with the best practices of software development was taken to make sure the thesis can be finished at all. Some of the tests were fixed where it was deemed simple to do so. Most of these changes were very mechanical though. The wordlength inference code also needed to be changed slightly to accept the new ranges as inputs from the type checker.

These changes also made the type checker aware of the range based semantics of integers in the wordlength inferrer – thus the type checker can be used to propagate type information about these ranges. This made all library functions which previously only used wordlengths work properly with arbitrary range. Users of the Spade language can now also specify ranges on their types themselves.

The very basic implementation in this thesis does not handle memories and registers. Though supporting them should be very minor work, but requires a lot more verification. There are also questions as to the language features of registers and memories should interact with the range-based syntax of integers. The unification rules from the type checker are also very strict and disallow things like passing a constant to a function that takes an integer argument that is not constant. How to solve these problems is according to the thesis authors a matter of taste, though the problems need addressing for a well functioning and friendly compiler.

Evaluating the Changes

After a change has been made it needs to be verified and tested. A custom version of the compiler with the changes described in Section 5.9 was made. Each program was synteshsed and PNRd 51 times for each of the three configurations AA, IA and AAIA – and a clean of the build environment was conducted between all steps. The changes breaks a lot of compatibility with older Spade-versions which severely limits the number of Spade programs that can be used since a dependency written for a different version of the compiler did not compile. After each build `swim clean` was run to remove build artifacts. The following programs were evaluated.

spade-memory-display

The small library `spade-memory-display` was used for evaluation. The changes to the compiler required changes to the library code in order to compile, this made it unclear how to compare directly with older versions of the compiler since small code changes can have large changes in the output. The synthesis PNRs were run for the ice40 architecture.

All builds produced the same metrics and is presented in Figure 5.10. The metrics shows no difference for any of the 153 place and route compilation of the spade-library. All of the compilations used 190 LUTs.

A Simple FIR-filter

FIR-filters are often implemented in hardware since they are of great use for signal processing in general. So a simple FIR-filter program with 40 elements with the type `int<0..100>` was setup[12]. This setup used the FPGA LFE5U-85F of the ecp5 architecture as target and slightly different nextpnr and yosys versions[13] than the previous tests.

Figure 5.11 shows the output from every PNR run for the FIR-filter program. All the outputs were identical regardless of what wordlength inference method was used. It is also worth noting that for IA and AAIA the return type of the `fir`-function can be simplified to `int<0..400000>`. The return type is required to be `int<-200000..400000>` for AA which goes into the negative even though it cannot possibly happen. The returned ranges are shown in Figure 5.12.

Comparing the Ranges For Simple Expressions

The wordlength inference was also compared on 3 expressions: $a-a+a-a+a-a$, $(a-b)\cdot(b-a)$ and $a\cdot a\cdot b\cdot b\cdot c\cdot c$, for all of the 3 methods AA, IA and AAIA. All arguments had a wordlength of 5 and of the range 0..100 the resulting range was noted in the table in Table 5.2.

	IA	AA	AAIA
		Input Wordlength = 5	
$a-a+a-a+a-a$	−93..93	0..0	0..0
$(a-b)\cdot(b-a)$	−961..961	−961..961	−961..961
$a\cdot a\cdot b\cdot b\cdot c\cdot c$	−15728640..16777216	−16794692..16794693	−15728640..16777216
		Input Range = 0..100	
$a-a+a-a+a-a$	−300..300	0..0	0..0
$(a-b)\cdot(b-a)$	−10000..10000	−2209..2209	−2209..2209
$a\cdot a\cdot b\cdot b\cdot c\cdot c$	$0..10^{12}$	$-9.6875^{12}..10^{12}$	$0..10^{12}$

Table 5.2: A table showing how different wordlength inference methods compare on simple mathematical expressions where all variables have the wordlength of 5 for the first three rows and all variables are in the range 0 to 100 for the last three rows. The leftmost column shows the expression that was wordlength inferred. No digits have been removed, rounded or truncated – though scientific notation has been used to compact the table for the final row.

Table 5.2 shows multiple different expressions had the wordlength inference algorithm applied to them. The inner cells in the table denote the evaluated range of the expression in the left most column according to each of the three methods outlined in the top column, all this for two different configurations of the input variables. Either all variables had the type `int<5>` so a, b and c had a given wordlength of 5 for the first three rows in the table, or the variables a, b and c had a type of `int<0..100>` for the bottom three lines in the table. From the table we can easily read out that AAIA always produced the tightest ranges. The multiplication expression ($a\cdot a\cdot b\cdot b\cdot c\cdot c$) shows an example expression where IA produces a tighter range than AA and why AAIA produces the tightest ranges in every case.

[12]The FIR-filter program in its entirety is available at https://github.com/FredTheDino/thesis-spade-lang/tree/main/messing/fir

[13]The nextpnr version was: `"nextpnr-ecp5" -- Next Generation Place and Route (Version nextpnr-0.6-30-g679b662a)`. The yosys version was: `Yosys 0.32+46 (git sha1 008b725c1, clang 10.0.0-4ubuntu1 -fPIC -Os)`

```
# Definitions of the operations that are constructed by the Visit function

# Consider each variable a mapping, where the value at index 0 denotes a constant
# and all other indices represent noise variables.
def Var(I, A..B):
  [0: (A + B) / 2, I: (A - B) / 2]

def RangeToAAF(A..B):
  # This operation is Opaque
  U = NewUniqueIndex()
  [0: (A + B) / 2, U: (A - B) / 2]

def Rad(A):
  # Sum all coefficients for noise variables - so everything except the constant
  # at index 0
  Sum(A[/0])

def Addition(A, B):
  [I: (A[I] or 0) + (B[I] or 0) for I in Union(Indices(A), Indices(B))]

def Multiplication(X, Y):
  # This step introduces a new noise variable
  P = RangeToAAF(Range::Multiplication(X[0]..X[0], Y[0]..Y[0]))
  # The swapped order of Y[0] and X[0] is not a mistake
  Affine(X, Y, Y[0], X[0], -P[0], (Rad(X) * Rad(Y)) + Rad(P))

def Affine(X, Y, Alpha, Beta, Gamma, Delta)
  Z = [I:   Alpha * (X[I] or 0)
          + Beta  * (Y[I] or 0)
          + (Gamma if I == 0 else 0)
        for I in Union(Indices(X), Indices(Y))]
  U = NewUniqueIndex()
  Z[U] = Delta
  Z

def Union(A, B):
  # This operation is Opaque
  RangeToAAF(Range::Union(ToRange(A), ToRange(B)))

def Negation(A):
  [I: -S for (I, S) in A]

def BitManip(A..B):
  # This operation is Opaque
  RangeToAAF(Range::Union(ToRange(A), ToRange(B)))

def ToRange(A):
  (A[0] - Rad(A))..(A[0] + Rad(A))

def NoInformation:
  pass
```

Figure 5.6: An outline of the different AA procedures, this explains how each constraint operation for the `Visit`-function should be evaluated if you want to use the AA method.

```
if InferMethod == OLD: return

# Here we call `Visit` (from the previous figure) internally to visit all the
# relevant expressions, we get # out a mapping from type variables to
# equations.
(Variables, Equations) = VisitAllIntegerExpressions(AST, TypeChecker)
# We ask the typesystem for its deduced sizes, since we have to agree with them
# and they are a good first guess. `Known` is a mapping from type variables to
# their known size, we will fill this in with more information as we proceed.
Known = ExtractKnownWordlengths(Variables, AST)

for _ in Equations:
  KnownAtStart = Known
  for (Var, Body) in Equations:
    OptionRange = case InferMethod of
                  # Call the evaluation of the equations using the appropriate
                  # method.
                  IA -> EvaluateUsingIa(Known, Body)
                  AA -> EvaluateUsingAa(Known, Body)
    match OptionRange:
      Some(Range) -> InjectAndCheckForContradictions(Range, Known)
      None -> pass
  # If we did not progress we abort
  if KnownAtStart == Known:
    break

for Var in Variables:
  unify(TypeChecker, Var, Known[Var])
```

Figure 5.7: The algorithm used in the seventh attempt to determine the wordlength for all expressions in a spade-program. `VisitAllIntegerExpressions` visits all the integer expressions in the AST with some help from the type checker state and returns a set of all type variables mentioned in an equation and a list of tuples containing type variables and the requirements posed from their wordlengths.

```
// Old Spade wordlength inference
fn add_and_subtract(x: int<5>) -> int<8> {
  (x - x) + (x - x) + sext(x - x)
}

// New Spade wordlength inference using AA
fn add_and_subtract(x: int<5>) -> int<0> {
  (x - x) + (x - x) + (x - x)
}

// New Spade wordlength inference using IA
fn add_and_subtract(x: int<5>) -> int<8> {
  (x - x) + (x - x) + (x - x)
}
```

Figure 5.8: A simple Spade function showing the difference in the wordlength inference with the new approaches with a focus on addition and subtraction.

```
fn twice<#A, #B, #M, #N>(x: int<A..B>) -> int<M..N> {
  2 * x
}

fn add_one<#A, #B, #M, #N>(x: int<A..B>) -> int<M..N> {
  x + 1
}

entity main(clk: clock, rst: bool) -> int<14..14> {
  // Does not compile, due to the inner function call returning an unknown type
  // let q: int<12..12> = twice(twice(3));

  let a: int<6..6> = twice(3);
  let b: int<7..7> = add_one(a);
  let c: int<14..14> = twice(b);
  c
}
```

Figure 5.9: A small Spade program that shows the new range syntax in context and gives an example of how integer range information is now more detailed.

```
ICESTORM_LC: 190/1280 (14.8%)
ICESTORM_PLL: 0/1      (0.0%)
ICESTORM_RAM: 0/16     (0.0%)
SB_GB: 2/8            (25.0%)
SB_IO: 4/112           (3.6%)
SB_WARMBOOT: 0/1       (0.0%)
```

Figure 5.10: The output from every place and route run given regardless of wordlength inference method for the spade-library spade-memory-display.

```
ALU54B: 0/78              (0.0%)
CLKDIVF: 0/4              (0.0%)
DCCA: 0/56                (0.0%)
DCSC: 0/2                 (0.0%)
DCUA: 0/2                 (0.0%)
DDRDLL: 0/4               (0.0%)
DLLDELD: 0/8              (0.0%)
DP16KD: 0/208             (0.0%)
DQSBUFM: 0/14             (0.0%)
DTR: 0/1                  (0.0%)
ECLKBRIDGECS: 0/2         (0.0%)
ECLKSYNCB: 0/10           (0.0%)
EHXPLLL: 0/4              (0.0%)
EXTREFB: 0/2              (0.0%)
GSR: 0/1                  (0.0%)
IOLOGIC: 0/224            (0.0%)
JTAGG: 0/1                (0.0%)
MULT18X18D: 40/156       (25.6%)
OSCG: 0/1                 (0.0%)
PCSCLKDIV: 0/2            (0.0%)
SEDGA: 0/1                (0.0%)
SIOLOGIC: 0/141           (0.0%)
TRELLIS_COMB: 2255/83640  (2.7%)
TRELLIS_ECLKBUF: 0/8      (0.0%)
TRELLIS_FF: 0/83640       (0.0%)
TRELLIS_IO: 342/365      (93.7%)
TRELLIS_RAMW: 0/10455     (0.0%)
USRMCLK: 0/1              (0.0%)
```

Figure 5.11: The output from every place and route run given regardless of wordlength inference method for the simple FIR-filter. The lines of most interest are `TRELLIS_COMB` and `MULT18X18D`. `TRELLIS_COMB` is the number of LUTs and stays constants. `MULT18X18D` is the number of multiplication circuits – which is equal to the number of multiplications in the filter so we know it is not fully optimized away.

- IA: `int<0..400000>`
- AA: `int<-200000..400000>`
- AAIA: `int<0..400000>`

Figure 5.12: The return types for the fir-filter test program where the wordlength inference method was varied.

Chapter 6

Discussion

After a thorough evaluation of the different suggested methods of implementing wordlength inference the implementations and their evaluations are discussed. Much of the discussion focuses on the final implementation since it produced the best results and was subject to the most evaluations.

6.1 Results

The Spade compiler was successfully changed to support ranges and more sophisticated wordlength inference. These changes were not able to be completed to the degree that they could be fully integrated into the mainline compiler – but most of the technical problems have been solved. What remains are language design decisions and glue code.

The table in Table 5.2 shows that the AAIA method results in the thightest ranges and the smallest wordlengths. That a combination of the techniques is superior is hardly surprising since IA handles multiplication better for expressions with a 0-lower bound while AA can handle subtractions of the same constants. A combination is bound to work better than any of them in isolation. Of special consideration is how IA handles 0 a lot better, since ranges bound by 0 are still bound by 0 even after multiplication, this is not the case for AA. This begs the question if there is a better method that would give even tighter ranges, and may be a suitable future work.

The internal changes to the Spade compiler typeinference caused a sea of troubles to appear. If the Spade project want to integrate these changes fully care needs to be taken when implementing and updating language features like memories. There is also a lot of work needed to port all the tests for the Spade language. All of these are fairly manual changes but may require a lot of time. To fully integrate these changes a series of language design decisions would also need to be taken, for example how constants passed to functions should type check or the example in Figure 5.9 which require extra help when inferring some type variables – this is believed to come from function calls being opaque to the wordlength inference

The opaqueness of functioncalls is a large problem in the current implementation, and perhaps the simplest way of solving is probably to move the wordlength inference into the typeinference module. The information that is needed is nowhere to be found in the wordlength inference stages of the compiler and should either be rebuilt or preferably given with the type checking state. But it may also be a sign that the method described in Section 5.6 might have been an even better choice – and if the typechecking were to be changed to function on ranges which was also required to change in Section 5.9 there may be a better implementation that easily fits into the typechecking step of the Spade compiler.

There was no discernible difference between the programs between synthesis and PNR runs with any of the wordlength inference methods. The table in Table 5.1 motivates this – even though the averages might suggest subtly against it. This means there is no real difference in LUTs or memories used when using the wordlength inference in this thesis. It may be claimed that the difference in variance is an improvement, it may also be due to random noise from the compiler since synthesis and PNR has undeterministic behavior. Figure 5.10 and Figure 5.11 support the claim of little difference. Though it is worth noting that something may be different in the compilation process since both of these figures show no noise. It may be the case that these improvements only aid the more sophisticated optimization steps in the synthesis and PNR-phase, and this could also explain

why the improvement negligible. This may signal an error in the experiments and that the Spade implementation with the range-based wordlength inference may have a bug which causes incorrect codegeneration, unfortunately there was no access to hardware to verify this implementation on. This is a major limitation of this study but does not deter from the wordlength inference presented or the conclusions reached.

The solver for wordlengths that is described in Section 5.8 is very simple – and as thus may not produce the best results possible. The solver does not handle solving equations with unknowns on both sides of the constraint, and variables being opaque to the AA method makes it produce worse results in some cases though most of the experiments in this work are simple enough to not make a huge difference. The inference algorithm could be modified to substitute entire expressions when evaluating using the AA method and it would probably produce slightly better results – though the types would still need to be reduced to ranges.

In the Spade language wordlength inference is now more flexible, and allows propagation of ranges between functions. That wordlength information can be sent in more detail between different parts of the program means that developers need fewer assumptions of how the code works. Less assumptions result in fewer sign-extensions and truncation operations on integer expressions. The power of AA also makes it beneficial to write larger expressions instead of splitting them up in smaller chunks with placeholder variables – since variables are always reduced to ranges by the wordlength inference algorithm for variable types.

Unfortunately the representation of integer values inside of the Spade compiler is ranges, which hinders some of the AA inference. Since each variable in Spade has a singular range paired with it the information of the sub-expressions is discarded: the information AA uses to function well is lost. For the AA wordlength inference method each variable is opaque and cannot be inspected into. Since variables are opaque it is better to inline expressions without the use of variables since the wordlength inference code can reason a lot better about those expressions.

Limitations in the Spade Compiler

The implementation offered in this thesis can have quite confusing error messages. This is due to the type checker discarding information of where type information came from – and the typeinference module is left looking at the expressions and types given by the type checker. Changing what the Spade compiler stores from the typechecking phase would require a fair bit of plumbing but still nothing hugely complicated to implement. For later stages in the compiler it would be preferable to make some changes in the compiler to support:

- A list of spans that have been unified to give a type for an expression – this would make it possible to point to the type signatures from stages that do e.g. constant folding or wordlength inference.
- A full list of the constraints and requirements for a type – currently the compiler tosses these constraints and requirements when they are deemed satisfied, this discards the sources of facts that both the type checker and wordlength inference module would find helpful.
- It would be great if `Requirement` and `ConstraintExpr` could be one construction – it would aid interoperability and remove some technical debt in the type checker.
- It would also be beneficial if each entity in the Spade program could be compiled as far as possible – so if type checking for one function of the program fails the wordlength could still be checked for an entity that is not related.

There are also profits in the area of usability of the Spade language – removing truncation operations from the Spade code can make the code a lot more readable. Since most of the truncations and sign extensions are required by the very simple wordlength inference method present before this thesis it is probably the single largest contribution that has been made. The change in wordlength inference makes the Spade compiler more flexible and makes it possible to remove a lot of the truncation and extension operations.

6.2 Method

The method does not focus directly on wordlength inference but focuses more on the software development side, which is in stark contrast with the research questions. This might however show more

fault in the usage of research questions than this thesis. The author do believe the code changes that came from this thesis to be among the best possible for the given time. The fumbling evaluation of alternative implementations gives a basis for that argument. It is however possible for such an implementation to have been found by thinking very carefully and very hard – though theoretically sound ideas often crumble in agony when faced with the sledgehammer that is reality. Especially if those theories are drawn from inexperience.

From software development it is well known that an iterative approach to software development often yields the best results in a fixed amount of time. The idea of using some form of agile development methodology seems to have been a sound idea.

It is also frustrating to leave the compiler in a state where most of the work is *almost* done. As stated earlier in the thesis – it is possible that some of the changes that need to be done are in fact harder than they appear. But programming work is well known to be hard to plan. This thesis does however leave a good base to work further on the compiler. It may also be for the better that these other changes are made by someone on the core Spade compiler team. Though this thesis has been a collaboration with the Spade compiler team, communication is always lossy, and details are bound to be forgotten. Hopefully these changes are well documented enough to be of use and be able to be well integrated into the language.

6.3 The Work in a Wider Context

More sophisticated wordlength inference in Spade unfortunately has little effect on the planet or the current war in Ukraine that is currently raging. But this contribution need not be discarded as useless. A slight improvement in efficiency for hardware designs can have cascading effects on how cheap it is to produce custom circuits. The cheaper cost can of course lead to an increased production which is what has already happened to goods such as computers and cars. Hardware description languages and FPGAs is used heavily in the weapons industry and thus code created in this thesis may contribute to more raging wars. However, it is essential to recognize that war is a double-edged sword – it can both preserve and condemn – yet its inherent nature remains one of destruction.

Hopefully hardware designers will rejoice over this work.

Chapter 7

Future Work

The implementation used in this thesis opens the door for an even more sophisticated approach where sub-expressions can be evaluated by either AA or IA. One can consider an evaluation tree where all possible combinations of AA and IA are used – though this tree may be too large to easily evaluate in practice, a technique like this would give theoretically optimal wordlengths for expressions. Removing the requirement that variables are opaque can also make the wordlength inference a lot better and might also be an interesting point of research in the future.

Since the Spade compiler now understands expressions and the values the expressions can be evaluated to it is trivial to check for expressions that evaluate to a constant. A constant expression may indicate a programmer error and can be raised to a warning. This range analysis can also be used to trim dead code even before the code generation, and then warn about unreachable paths.

Since the compiler now reasons about ranges of integer values unsigned integer optimizations should be possible to do in a multitude of places. Maybe even automatically generating code that uses unsigned integers if the code never goes below 0. But this can of course be taken a step further, and all expressions can be re-written by the compiler to avoid going below 0. We can trivially rewrite the expressions inside the compiler since integer arithmetic is a well behaved ring (unlike floating point numbers). This can lead to improved resource usage in some places. Maybe it can even be beneficial to shift the representation of numbers inside the compiler.

Some of the typing and inference rules need to be updated to make the language work well with e.g. constants passed to functions. Looking into these kinds of rules can make the language clearer and more expressive – pairing it with a usability study could be very enlightening.

There are also more advanced wordlength inference methods. There are alternatives to AA which may prove useful – like ME-gPC. Maybe it is possible to track every possible integer value an expression can take, and allowing these holes in the expressions would make it possible to give even more precise guidance and help.

Maybe it can be possible to pair the Spade-language with more complex formal verification methods, and allow formal verification of circuits in the language directly. The language for the generics can also be extended to support this formal verification step, so that constraints for functions that require certain ranges based on the input can be wordlength inferred and type checked more seamlessly. For an example envision the type signature `fn double<#A, #B>(a: int<#A..#B>) -> int<2*#A, 2*#B>`.

This work has added a separate wordlength inference module in addition to the type checking, but it may still be very beneficial if they are one and the same – and thus looking into more complex type systems or reworking the current typesystem may still be a good idea, this is another way to make function calls less opaque.

The level of improvement that is gained by this improved wordlength inference may be overshadowed by the work of the synthesis and PNR tools – as such it may be the case that we alleviate a bit of syntehsis and PNR work with more improved wordlength inference, this may lead to better compile-times. Investigating the relationship between compile-times and these kinds of optimizations may be very interesting.

Chapter 8

Conclusion

Programming languages are complex things – but they need to be complex to express complex and precise ideas. The wordlength inference outlined in this thesis shows a good way to extend the Spade hardware description language to more flexibly handle wordlengths. The wordlength inference works well with the current Damas-Hindley-Milner type checker and will allow language designers of the Spade language to make an active choice in how the Spade language should interact with wordlengths. There was no difference in the resource usages of Spade programs before or after the wordlength inference from this work was introduced. The authors believe that the real benefit from these changes will not come from performance gains – though better performance may be possible in the future – but the "soft" value of expressiveness and clearer programs. A possibility of clearer intent will make it possible for the Spade compiler to more clearly understand the programmer and may make more readable code and aid optimizations. We will also answer the research questions posed in Section 1.2.

8.1 How can interval arithmetic and affine arithmetic be used to implement wordlength inference?

The most optimal way was to combine both interval arithmetic and affine arithmetic. Since both have different strengths and weaknesses the combination of the methods costs little in the way of resources but can in some instances give a lot smaller ranges which leads to smaller wordlengths. If this work is to be extended to support unsigned integers this combined method is bound to come in very handy.

8.2 How does wordlength inference and optimization affect the number of LUTs?

This thesis clearly shows that this implementation of wordlength inference has no discernible effect on the number of LUTs in a circuit. If there are gains here they are small or can also be gained from running the same synthesis and PNR operation multiple times and picking the best one. This study can be based on to introduce unsigned integers into the Spade language - which may give more substantial gains.

8.3 Can wordlength inference be used to create more reusable code?

The final state of the changes from this thesis made the compiler very finicky about wordlengths. The expressiveness in the typesystem has increased which has made it possible to express some types more clearly and making certain function types more expressive, making functions a more well integrated language construction – aiding the reusability of code. Though it does not aid the resuability greatly the authors claim this is an improvement no matter how modest it is. Future work could easily expand on this.

Bibliography

[1] July 2023. URL: https://en.wikipedia.org/wiki/Computer_(occupation).

[2] Frans Skarman et al. *Spade*. DOI: 10.5281/zenodo.7713114. URL: https://zenodo.org/records/8279534.

[3] Frans Skarman and Oscar Gustafsson. "Spade: An HDL inspired by modern software languages." In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)* (2022). DOI: 10.1109/fpl57034.2022.00075.

[4] Jorge Stol and Luiz Henrique De Figueiredo. "Self-validated numerical methods and applications." In: *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer*. Vol. 5. 1. Citeseer. 1997.

[5] Xiaoliang Wan and George Em Karniadakis. "An adaptive multi-element generalized polynomial chaos method for stochastic differential equations." In: *Journal of Computational Physics* 209.2 (2005), pp. 617–642. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2005.03.023. URL: https://www.sciencedirect.com/science/article/pii/S0021999105001919.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques and tools*. Addison-Wesley, 1985.

[7] About Robert Nystrom. *Crafting interpreters*. URL: https://craftinginterpreters.com/.

[8] *TDDB44 lectures and slides*. 2022. URL: https://www.ida.liu.se/~TDDB44/lectures/lectures.en.shtml.

[9] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[10] Luis Damas and Robin Milner. "Principal type-schemes for functional programs." In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82* (1982). DOI: 10.1145/582153.582176.

[11] W. A. Howard and H. B. Curry. *THE FORMULAE-AS-TYPES NOTION OF CONSTRUCTION*. 1980. URL: https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf.

[12] *Generic Data Types - The Rust Programming Language*. June 2023. URL: https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics.

[13] Nov. 2023. URL: https://coq.inria.fr/refman/language/cic.html.

[14] Nov. 2023. URL: https://agda.readthedocs.io/en/v2.6.0.1/getting-started/what-is-agda.html.

[15] Nov. 2023. URL: https://elm-lang.org/.

[16] Aug. 2023. URL: https://gitlab.com/spade-lang/spade/-/blob/master/ARCHITECTURE.md.

[17] The Rust Foundation. *Rust Programming Language*. July 2023. URL: https://www.rust-lang.org/.

[18] Ian Kuon, Russell Tessier, and Jonathan Rose. “FPGA Architecture: Survey and Challenges.” In: *Foundations and Trends in Electronic Design Automation* 2.2 (2008), pp. 135–253. ISSN: 1551-3939. DOI: 10.1561/1000000005.

[19] *FPGA for Military Applications - Intelő FPGA*. June 2023. URL: https://www.intel.com/content/www/us/en/government/products/programmable/applications.html.

[20] N. Doi et al. “Minimization of fractional wordlength on fixed-point conversion for high-level synthesis.” In: *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)* (2004). DOI: 10.1109/aspdac.2004.1337544.

[21] Yohann Uguen. “High-level synthesis and arithmetic optimizations.” Theses. Université de Lyon, Nov. 2019. URL: https://hal.science/tel-02420901.

[22] Jan Frigo, Maya Gokhale, and Dominique Lavenier. “Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective.” In: *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. FPGA ’01. Monterey, California, USA: Association for Computing Machinery, 2001, pp. 134–140. ISBN: 1581133413. DOI: 10.1145/360276.360326. URL: https://doi.org/10.1145/360276.360326.

[23] Jana Dunfield and Neelakantan R. Krishnaswami. “Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism.” In: *Int’l Conf. Functional Programming*. arXiv:1306.6032[cs.PL]. Sept. 2013.

[24] Pekka Abrahamsson et al. *Agile Software Development Methods: Review and Analysis*. 2017. arXiv: 1709.08439 [cs.SE].

Appendix A

Versions and Hashes

This section contains githashes of the Spade compiler that have been mentioned in this thesis. All code is available at https://gitlab.com/FredTheDino/spade – the thesis GitLab.

```
branchname: the-simplest-implementation
git-commit: 74c966a0317aa738017d2edf15def4719fe8dc95

branchname: the-second-simplest-thing
git-commit: 3d92c0e4b28b64104f700c3d299f62e7938cb016

branchname: the-third-simplest-with-equations
git-commit: 83d1da48f5010767b9a96aea0a5bca13b7415084

branchname: the-fourth-attempt-now-with-equivelence
git-commit: ba0fa1baab56a725c31f703204da3b7d5f44380a

branchname: the-fifth-attempt-now-without-returns
git-commit: d8e9acbca755dea0c2ee78a014c578064c07d47e

branchname: the-sixth-attempt
git-commit: 11b3d060bc34145fda4918d255b186cb4057f07f

branchname: the-seventh-attempt-almost-as-simple-as-attempt-one
git-commit: 889afd61a59f04f60730964b8ae7a2703110dd99
url: https://gitlab.com/spade-lang/spade/-/merge_requests/200


branchname: wordlength-inference/push-the-changes-further
git-commit: ee9980c6ec518dbdf0794dbb9193c5b1c9b6945e
url: https://gitlab.com/spade-lang/spade/-/merge_requests/208
```

Appendix B

Notes and Curiosities Found While Writing

When reviewing the code for this work an error in the original code for the second attempt discussed in Section 5.4 was found. The function `check_expr_for_replacement` should refer to `BitsToRange` on line 1036 – the changes presented in this section never compiled or worked as intended. Not that the code needed to work since a proper typesystem can catch a lot of erroneous implementations before they even run.