TYPE CHECKING IMPLEMENTATION IN SCALANESS/NEST

A Thesis Presented

by

Michael P. Watson

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fullfillment of the Requirements
for the Degree of Master of Science
Specializing in Computer Science

May, 2014

Accepted by the Faculty of the Graduate College, The University of Vermont, in partial fulfillment of the requirements for the degree of Master of Science, specializing in Computer Science.

Thesis Examination Committee:

________________________________ Advisor
Christian Skalka, Ph.D.

____
Alan Ling, Ph.D.

________________________________ Chairperson
Jeffery Dinitz, Ph.D.

________________________________ Dean, Graduate College
Cynthia J. Forehand, Ph.D.

Date: March 26, 2014

Abstract

Programming wireless sensor networks is a notoriously difficult task due to severe resource constraints at the sensor node level. One approach to this issue, *Scalaness/nesT*, uses staged programming to offload computations from the sensor node to a more powerful base station and subsequently specialize the sensor network code with the results. Scalaness and nesT are implemented versions of DScalaness and DnesT, two foundational languages that were presented with a well-founded type theory based on the principle of *cross-staged type safety*. This thesis details the implementation of the DScalaness/DnesT type theory into the actual Scalaness/nesT languages, focusing on how the implemented type checking system maintains the strict type principles set forth in the foundational languages. The first stage language Scalaness is an enhanced version of the object oriented programming language Scala, and its type checking system is implemented through direct modification of the Scala compiler. The second stage language nesT is implemented as a type safe dialect of sensor network programming language nesC, and its type checking system was built from the ground up. Through exploration of the type checking implementation details of both the first and second stage languages, this thesis provides a fully functional demonstration of the principles of cross-stage type safety.

Acknowledgments

I would like to acknowledge a number of people without whom this thesis never would have been possible. First, I'd like to thank the faculty and staff of the Math and Computer Science departments at the University of Vermont for all of their support over the years, especially my advisor Christian Skalka who provided me with the opportunity to work on this project, for which I am incredibly grateful. I'd also like to thank my friends and family who stuck with me and had unwavering belief in me throughout my education. Finally, I owe an enormorous debt of gratitude to Peter Chapin, whose advice and support were invaluable throughout my entire time on this project.

Table of Contents

List of Figures

Chapter 1

Introduction

Programming on wireless sensor networks is a notoriously difficult task due to constrained resources at the sensor node level. These sensor nodes, or motes, are limited in RAM, ROM, and basic processing power, which can provide several challenges to programmers who wish to design applications to be run at the mote level. Researchers have developed many approaches to deal with the issue of limited resources at the mote level, one of which is using staged programming (Consel, Hornof, Marlet, Muller, Thibault, Volanschi, Lawall, and Noyé 1998; Taha 2004; Taha and Sheard 1997) to shift some of the computation done on the node to an earlier stage of compilation (done on a more powerful system).

One of these staged programming systems is DScalaness/DnesT (Chapin, Skalka, Smith, and Watson 2013), a foundational language designed with two major goals: to move intensive computations off of the node by executing them during the first stage and then specializing the second stage code with their result, and to achieve so-called "*cross-staged type safety*". Cross-staged type safety is a principle that was first introduced in the conceptual language <ML> (Liu, Skalka, and Smith 2011). This principle states that if first stage code that generates second stage code is well-typed, then the second stage code it generates necessarily must be well-typed as well, and therefore enjoy the benefits of type safety. DScalaness and DnesT were presented (Chapin, Skalka, Smith, and Watson 2013) with a foundational program syntax, as well as a number of type rules and principles designed to serve as a basis for a fully implemented language. This implemented language came in the form of Scalaness and nesT (Chapin 2014). The first stage language Scalaness is a modified version of Scala (Odersky, Spoon, and Venners 2011), which is designed to specialize code for the second stage language nesT, a reduced version of nesC (Gay, Levis, von Behren, Welsh, Brewer, and Culler 2003) with additional features and type analysis.

Within the first stage Scalaness program, second stage code is manipulated through the use of objects called *nesT Modules*. These modules contain components of second stage nesT code and are treated as first class values in Scalaness. The goal of a Scalaness program is to specialize these components of nesT code by performing computations on a more powerful base station that would otherwise need to be performed on the nodes, and inserting the result of these computations into the existing nesT code. This specialization occurs during the run time execution of Scalaness code, which may occur either in a lab or on a mobile computing device in the field. (See Fig. 1.1) This specialization modifies the nesT code, but because of the principle of cross-stage type safety, any specialized code that is produced by execution of a Scalaness program is expected to be type safe. This is achieved by type checking the code components of nesT Modules during the compilation of the first stage program. Though the nesT code has yet to be specialized when it is type checked, the theory behind the DScalaness/DnesT type checking system is designed to state that the program is well-typed if and only if every possible specialization of the code would be considered well-typed. Thus, execution of any Scalaness program that is approved by Scalaness type checking will necessarily produce well-typed specialized nesC code, thus implementing the principle of cross-stage type safety. This thesis focuses on the implementation of the type checking system for both Scalaness and nesT, and how the implementation maintains the strong foundational type principles set forth by DScalaness

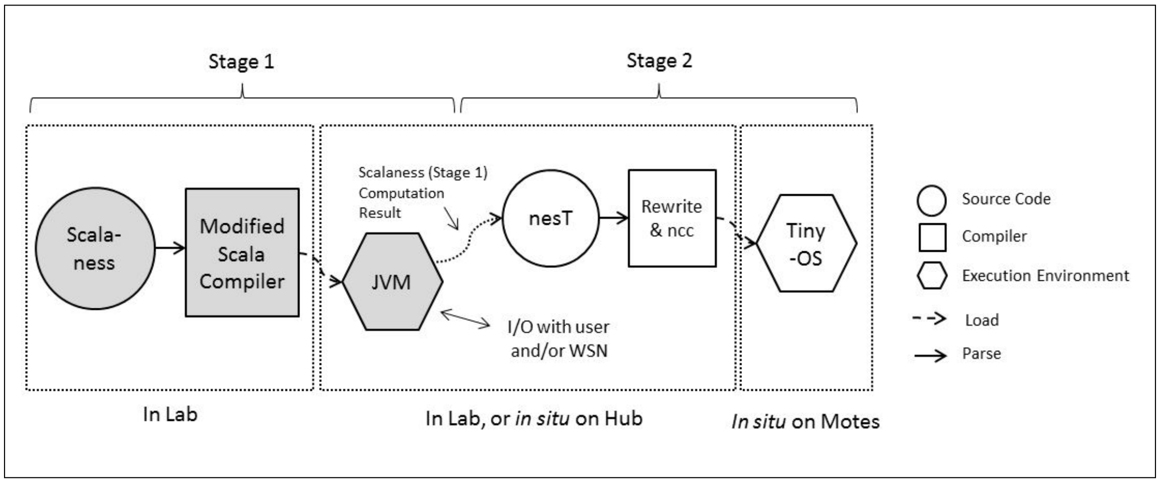

Figure 1.1: Scalaness/nesT Compilation and Execution Model

and DnesT.

1.1 Thesis Organization

The content of this thesis is generally split into two main categories. First, the technical core of the thesis, which explains the transformation from the foundational DScalaness/DnesT languages into an actual working type system, as well as the motivation behind the individual design decisions and how they reflect the original theory. Second, the discussion of how the type checking system was designed from a software engineering perspective, which explains all of the components required to support a multi-staged type checking system. Both are explained in detail throughout the course of the thesis, combining to paint a full picture of how the type checking system was built, and how it implements the type principles of DScalaness and DnesT.

In Chapter 2, the implementation of the nesT type checking system is presented. The technical core of this chapter is the motivation and justification behind the transformation from the foundational DnesT type system into the actual implemented nesT. This transformation is explained via presentation of the original DnesT syntax, subtyping relation, and type rules, along with their nesT equivalents. Additionally, this chapter includes sections on the various constructs required to create a fully operational type checker for nesT, including the nesT abstract syntax tree (AST), the symbol decoration process, and additional nesT run time support.

Chapter 3 presents the Scalaness type checking system. This chapter focuses on how the Scala typer was extended (via direct modification of the Scala compiler) to create a type checking system for Scalaness. This chapter explains the complicated nature of the Scala compiler and the various work required to extend it to allow for the typing of nesT module types and operations. The technical core of this chapter is the transformation from the DScalaness type rules (which type the primitive module operations - composition, instantiation, and imaging) into their Scalaness equivalents. Like in Chapter 2, this is achieved through presentation of the original DScalaness type rules and their Scalaness equivalent, along with explanation of how the Scalaness rules properly implement the DScalaness theory.

Finally, Chapter 4 provides detail on additional Scalaness level support. This includes an explanation of how Scalaness Type Abbreviations were implemented, and why they are an important tool for coding in Scalaness. Additionally, this chapter presents a specification for the syntactical transformation from DScalaness to Scalaness, which in combination with the type rule implementation from Chapter 3, paints a full picture of how DScalaness has been implemented.

1.2 Thesis Contributions

The goal of this thesis was to prove that an implemented version of DScalaness/DnesT was possible, thus creating a working demonstration of cross-stage type safety. This implementation was achieved in the form of Scalaness and nesT. The implemented type checking system spans two stages, including a full nesT type checker that was built from scratch specifically for the purpose of type checking components of code within nesT Modules. The type checking algorithm was implemented in direct accordance with the DnesT type rules, thus maintaining the foundational type theory set forth by that language. In addition to implementing the type rules, an entire support network was built around the type checking algorithm to ensure that type checking was possible, including symbol tables, a declaration parser, and implementations of the DnesT subtyping and promotion relations. Additionally, all DnesT types were given nesT equivalents, contributing to a fully working nesT type checker that is able to make correct type judgments on components of nesT code.

Additionally, the Scalaness type checking system was successfully built via direct modification of the Scala compiler. Multiple Scalaness constructs were built from scratch and added into the compiler to support this. These include nesT Module types and operations, Scalaness Type Annotations, and Scalaness Type Abbreviations. Because of the firm DScalaness/DnesT theory that this type system was based on, the completed languages were not only able to specialize code at Scalaness run time, but also perform accurate type analysis during the compile time of the first stage. The Scalaness compiler was able to reject nesT code that would be considered unsafe upon specialization, and of course reject any Scalaness or nesT code that was unsafe on its own. The finished system was used on a series of small/medium scale examples, and even one large example (Chapin 2014), to illustrate that the theory of DScalaness/DnesT had been accurately implemented. Thus, the completed system creates a working demonstration of a staged programming language which lessens the burden of computation on the sensor nodes without sacrificing a firm type theory.

1.3 State of Implementation and Code Repository

Scalaness and nesT have both been implemented to a fully working level, and are available publicly at `https : //github.com/pchapin/scala`. This repository contains all of the source code required to build a working version of the Scalaness compiler. With the exception of the "Scalaness" folder, all of the files and paths in the main repository come from the original Scala compiler, but have modified or added code and files which make up both the Scalaness compiler and the nesT compiler. The previously mentioned Scalaness folder is full of both small and large examples that run on a fully built version of the Scalaness compiler and, once executed, output specialized versions of nesC code. Within the files and folders of the Scala compiler itself, any path that begins with `/uvm/edu` is code exclusive to the Scalaness and nesT compiler. For example, the majority of code for the nesT compiler is located within the `scala/src/library/edu/uvm/nest` path. More specifics can be found in README files spread throughout the Scalaness repository, as well as instructions on how to build the Scalaness compiler for actual use. Additionally, a guided tour of Scalaness/nesT including the complete compiler source code as well as compile ready examples and demonstrations of type errors was collected by Peter Chapin and can be found at `http://tinyurl.com/a85z8cu`.

Chapter 2

nesT Typechecking

This chapter covers the theory and implementation details behind the type checking system of the second stage language, nesT. nesT implements the theoretical language DnesT (Chapin, Skalka, Smith, and Watson 2013) and is designed for actual use as a sensor network programming language. The DnesT language was presented with a complete syntax (Fig. 2.1) and a number of type rules (Fig. 2.8) designed to serve as the foundation for a full language. nesT is the practical implementation, which implements the syntax and expands the type rules into a fully functional well-typed language. nesT code is written in individual files, which are the basis for nesT Modules - objects that are manipulated as first class values in the first stage Scalaness language. When a new nesT module is introduced in Scalaness, the nesT code is compiled by first parsing the language into an easy to manipulate AST, decorating the symbol table, and then running the type checker. All of these stages of compilation have been built from scratch, and this chapter will focus on the type checker, and how the design properly implements the strict type theory of DnesT. The result of the nesT type checking system is a type result that is decided on pre-specialized nesT code, but is expected to hold after specialization of the code, thus satisfying the principle of cross-stage type safety. This chapter will also explain additional support for nesT level type checking, including decoration of the symbol table and run-time array bounds checks.

$$
\begin{array}{lll}
\varsigma, \tau & ::= t \mid \top \mid \mathbf{uint8} \mid \mathbf{uint16} \mid \mathbf{uint} \mid \mathbf{uninit} \mid & \textit{types} \\
 & \quad \{\bar{\mathsf{l}} : \overline{\tau}\} \mid \tau[\,] \mid \tau\star \mid (\tau) & \\
e & ::= v \mid \ell e \mid e\ \mathit{op}\ e \mid (\tau)\, e \mid \mathsf{f}() \mid \ell e = e \mid \&\ell e \mid & \textit{expressions} \\
 & \quad \mathtt{\textbf{if}}\ (e)\ e\ \mathtt{\textbf{else}}\ e \mid \mathtt{\textbf{while}}\ (e)\ e \mid e; e \mid \mathbf{post}\ \mathsf{f}() & \\
\ell e & ::= x \mid e[e] \mid e.\mathsf{l} \mid \star e & \textit{l-values} \\
v & ::= n^{8} \mid n^{16} \mid \mathbf{uninit} & \textit{base values} \\
\mathit{op} & ::= + \mid \star \mid \&\& \mid == \ldots & \textit{operations} \\
\mathit{id} & ::= \mathsf{f} \mid x & \textit{identifiers} \\
d & ::= \tau\ x\ =\ e \mid \tau\ x\ =\ [\mathsf{|}\,\overline{e}\,\mathsf{|}] \mid \tau\ x\ =\ \{\bar{\mathsf{l}} = \overline{e}\} \mid \mathsf{f} : \tau = (e) & \textit{declarations} \\
 & & \\
T & ::= \bar{t} \preccurlyeq \overline{\tau} \qquad V ::= \overline{x} : \overline{\tau} & \textit{type and value parameters} \\
\iota & ::= \bar{\mathsf{f}} : \overline{\tau} \qquad \xi ::= \bar{\mathsf{f}} : \overline{\tau} = \overline{(e)} & \textit{imports and exports} \\
\varepsilon & ::= \bar{\mathsf{f}} : \overline{\tau} & \textit{export signatures} \\
\mu & ::= <T; V>\{\iota; \overline{d}; \xi\} & \textit{module definitions} \\
\mu\tau & ::= <T; V>\{\iota; \varepsilon\} & \textit{module signatures}
\end{array}
$$

Figure 2.1: Program Syntax of DnesT

2.1 The Type Checker

This section describes the function and implementation of the nesT type checker. Type checking is a vital component in the compiler of any programming language, and this is especially true in nesT, which is based on the firm type principles of the DnesT language. In general, type checking occurs during compilation of the code in order to detect and prevent errors that may appear during the run time of a program. Thus, type checking is considered a preventive measure, and must approve any code before it can be executed at run time. The type checker accomplishes this by examining the code and determining whether or not it is valid with respect to its type usage. Specifically, the type checker will examine constructs like expressions and statements to determine if the arguments and the return types are in line with what the compiler expects. For example, if a function is defined to accept an array of characters (`Array[Char]`) as its argument, but is called using an array of integers

(`Array[Integer]`), the type checker would be able to identify this "type error" and alert the user, usually bringing the compilation to a halt.

Of course, not every type checking case is that simple, and because of this, type theory in a well founded language consists of a set of type rules that define how a given expression or statement should be handled. If designed correctly, proper implementation of these type rules should result in the identification of all type errors that pass through the type checker. A number of DnesT level type rules were presented in the DScalaness/DnesT paper (Chapin, Skalka, Smith, and Watson 2013) and in conjunction with several new type rules, served as the basis for the type theory that was implemented in the nesT type checker. The type checking algorithm (described in detail in Sect. 2.1.7), functions by traveling through the AST of the code, looking for expressions and statements, and then executing the logic derived from the corresponding type rule. This results in a type checking system that exhaustively analyzes the code and creates a "pass or fail" result based on a series of well founded type principles. In nesT this type result is determined before specialization of the code, but due to the DnesT type theory that it was implemented from, this same type result is expected to hold even after specialization of the code is completed.

2.1.1 The Abstract Syntax Tree

Before talking about how the type checking algorithm works, it is important to understand how the code appears to the compiler after the parsing phase. After the nesT code is parsed, an Abstract Syntax Tree (AST) is created that is populated by AST Nodes that represent each component of the nesT language (expressions, constants, variables, etc.). Traversing through the AST is simple due to fields in the AST node that reference not only the children of a given node, but also its parent. Additional fields in the AST Node point to its Token Type (a constant number that corresponds to what the AST node represents - such as 41

```
STATEMENT (133)
  = (11)
    POSTFIX_EXPRESSION (113)
      addResult (119)
    + (108)
      POSTFIX_EXPRESSION (113)
        1 (29)
      POSTFIX_EXPRESSION (113)
        2 (29)
```

Figure 2.2: AST Tree of a simple operation: `addResult` $= 1 + 2$.

for a STRING or 30 for an IF statement), the text of the node (useful in the case of a string or numerical value), and the Symbol Table that is associated with that node (more in Sect. 2.2.3). These variable fields, as well as the ability to traverse up and down the AST, lend heavily to the inspiration behind the type checking algorithm. An example section of the AST is shown in Fig. 2.2, where there numbers next to each node represent their Token Type and indentation represents the depth of the tree.

2.1.2 nesT Type Representations

In general, a type checker functions by assigning certain objects (such as variables, values, and expressions) a given type that represents the structure or use of the object. In the case of nesT, every AST node in the parsed code will be assigned a type by the type checker, therefore it is important to have a variety of types that represent every possible component of the code. The DnesT syntax (Fig. 2.1) provides a series of types τ which can be used to describe every value in the DnesT language. In nesT, these DnesT types exist as *nesT Type Representations*, which can come in various forms - all based on their DnesT equivalents. These include a series of primitive types which represent the most simple of the language components, as well as more structured types that build off the primitive types, such as functions and arrays.

In DnesT (Fig. 2.1), the primitive integer type has three representations, `uint8`, `uint16`, and `uint`. As shown in Fig. 2.3, nesT expands these types into six separate integer types that represent three different sizes of either the signed or unsigned variety. The default type representation that is assigned to any constant is based on the size of the constant, and is always defaulted to a signed integer. Thus, a small constant in the code may be assigned a type representation of `Int8` in nesT, while a larger constant may be designated an `Int32`.

$$
\begin{array}{lcl}
\mathtt{t} & ::= & \mathtt{Int8} \mid \mathtt{Int16} \mid \mathtt{Int32} \mid \mathtt{UInt8} \mid \mathtt{UInt16} \mid \mathtt{UInt32} \mid \mathtt{Char} \mid \mathtt{Okay} \mid \\
 & & \mathtt{Top} \mid \mathtt{ErrorT} \mid \mathtt{Pointer(t)} \mid \mathtt{Array(t,s)} \mid \mathtt{Function(t,List[t])} \mid \\
 & & \mathtt{Structure(t,List[(s,t)])} \mid \mathtt{Uninit} \mid \chi \\
\chi & ::= & \mathtt{TypeVariable(s)} \\
\mathtt{mt} & ::= & \mathtt{Module(List[(}\chi\mathtt{,t)],List[(s,t)],List[(s,t)],List[(s,t)])}
\end{array}
$$

Where `s` *is a string.*

Figure 2.3: nesT Type Representations

Building off of the primitive types in DnesT are a number of structured types, which are assigned to the more expressive components of the language. Types like arrays ($\tau[]$), pointers ($\tau*$), functions ((τ)), and structures ($\{\bar{\mathsf{l}} : \bar{\tau}\}$) build off the primitive types to denote more intricate components of the language. Each of these DnesT types was given a corresponding type representation in the nesT implementation, such as arrays (`Array[t]`), pointers, (`Pointer[t]`), and structures (`Structure[(String, List[(String, t)])`). DnesT function types were also implemented in nesT, but were expanded from the original type to include not only the return type of the function but any parameters types as well, as shown in their nesT type representation (`Function[(t, List(t))]`).

Additionally, there is a type representation used for the nesT level typing of nesT Modules. This module type is assigned to the root node based on the complete typing of the code and is the last type assigned during type checking. nesT Module Type Representations are never passed around or used in nesT typing, but rather are types that are assigned to the code as a whole, so that the components of nesT code may be manipulated at the Scalaness level. nesT Module types are explained more in Sect. 3.1.

2.1.3 nesT Type Variables and Coercions

The final nesT Type Representation is known as a Type Variable. These Type Variables exist in the code essentially as a placeholder type, and are associated with an assigned upper bound type for purposes of type checking. Type Variables are declared as a parameter of an un-instantiated nesT Module, and are not resolved until instantiation of the Module with a concrete type (usually decided at run time of the first stage code). Thus, when nesT Type Checking occurs (during compilation of the first stage program), these Type Variables exists simply as variables, and thus must be treated by the type checker in a special way.

Each Type Variable is of the form `TypeVariable[String]` where the String that is used as a parameter to the Type Variable is what represents this type in the nesT code. For example, a `TypeVariable[MyType]` upper bounded by the concrete type `Int32` could be used as a type to declare a variable in the nesT code (`MyType x = 12;`).

The use of Type Variables is how DnesT (and subsequently its nesT implementation) is able to type check pre-specialized code. Because Type Variables are given upper bounds, which the specialized value will always satisfy, the type checker is able to accurately render type analysis on a value whose type may be undecided at compile time. This means any types decided during the run time of the first stage program that generates the code are expected to maintain the type result decided by the nesT type checking system.

A special section of the symbol table (described further in Sect. 2.2.3) is dedicated to storing information about these Type Variables and their upper bounds. These tables take the form of Scala maps, and map a Type Variable object to a nesT Type Representation (which in this case signifies an upper bound). These Type Variable symbol tables are nesT implemented versions of a DnesT "Type Coercion" (*T* in Fig. 2.4). In DnesT, a Type Coercion contains information about these placeholder Type Variables and their upper bound relations, whether they be to concrete Type Representations or even other Type Variables. These Type Coercions then aid in passing judgment on whether a typing relation that contains a reference to a Type Variable is valid. Thus, these Type Coercions serve as additional information to complete a set of standard rules designed to allow or disallow given statements and expressions in the nesT language.

2.1.4 nesT Subtyping

$$\textsc{ReflS}\quad T \vdash \tau \preccurlyeq \tau \qquad \textsc{TopS}\quad T \vdash \tau \preccurlyeq \top \qquad \textsc{TransS}\quad \frac{T \vdash T(t) \preccurlyeq \tau}{T \vdash t \preccurlyeq \tau} \qquad \textsc{UintS}\quad T \vdash \mathbf{uint8} \preccurlyeq \mathbf{uint16} \preccurlyeq \mathbf{uint}$$

$$\textsc{FnBodyS}\quad \frac{T \vdash \tau_1 \preccurlyeq \tau_2}{T \vdash (\tau_1) \preccurlyeq (\tau_2)} \qquad \textsc{StructS}\quad \frac{T \vdash \overline{\tau_1 \preccurlyeq \tau_3}}{T \vdash \{\overline{\mathsf{l}_1 : \tau_1} \uplus \overline{\mathsf{l}_2 : \tau_2}\} \preccurlyeq \{\overline{\mathsf{l}_1 : \tau_3}\}}$$

Figure 2.4: Subtyping Rules

One of the most significant contributions of the DScalaness/DnesT system is the use of subtyping at the DnesT level. Subtyping allows more freedom to the programmer throughout the code, and can also be used to substitute types with a smaller memory footprint where a larger type was expected, thus saving space in the resource constrained mote environment where the code will run. For example, suppose a Function is defined to accept a 16 bit integer as a parameter, however, the constant used to call that function is small enough to be represented in 8 bits. Because `int8` $\preccurlyeq$ `int16`, we can pass the Function an 8 bit integer, which it will accept. This prevents the need to declare the integer as an `int16` type, thus saving a full 8 bits of memory. This idea of subtyping was brought to nesT via implementation of the DnesT subtyping relation (Fig. 2.4), which is in turn based on previous foundational work in subtyping (Liu, Skalka, and Smith 2011; Ghelli and Pierce 1998).

The real power of the subtyping relation comes in conjunction with the use of Type Variables throughout the code. During type checking, Type Variables have yet to be resolved and therefore can be type checked only by their upper bound. Because of this, a Type Variable may take many different forms when it is actually instantiated, but the type checker must know whether to allow or disallow the use of Type Variables in different spots in the code. Using the subtyping relation and knowledge of the Type Variables' upper bounds, decisions can be made on whether or not a type is valid in a certain spot given any of its possible instantiations. If a Type Variable's upper bound is a subtype of the type required by a typing rule, it can be confirmed that any instantiation of that Type Variable (assuming it is equal to or a subtype of its upper bound) will be valid in that given spot. This is instrumental to the result of nesT, a language that can be confirmed to be type correct even before it is fully specialized.

The subtyping relation of any two types is decided by a method written at the nesT level called `areSubtypes()`. This method implements the subtyping relation that was put forth in DnesT (Fig. 2.4). The `areSubtypes()` method is called with three arguments, the Type Coercion *T* (called `typeVars` in the nesT code), and the left and right sides of a subtyping relation (called `leftType` & `rightType`). The result of the `areSubtypes()` method call is a boolean, returning `true` if and only if the `leftType` is considered a subtype of the `rightType` based on the judgment decided by the subtyping rules and the Type Coercion.

2.1.5 Subtyping Implementation: Case Analyses

The implementation of the `areSubtypes()` is a direct translation from the DnesT subtyping relation into the nesT compiler. To achieve this, each case of the subtyping relation (Fig. 2.4) was implemented as a pattern match case in the `areSubtypes()` method.

The trivial subtyping rules are implemented into `areSubtypes()` in the simplest of forms, as each call to `areSubtypes()` begins with an equality check that covers both the REFLS rule for equality as well as the TOPS global supertype:

```
if (left == right || right == Top) true
```

One of the basic subtyping rules is the UINTS rule, which simply says that a smaller sized integer is a subtype of a larger sized integer. This is implemented in `areSubtypes()` with both unsigned and signed integers, with the left side being matched to the right side in a multitude of cases. The following case shows the comparison made if the left type is a `UInt8`:

```
case UInt8 => right match {
            case UInt16 | UInt32 => true
            case _  => false
          }
```

This is then followed by a complete set of cases that allow for any combination of left side and right side integers.

```
val typeCase = (left, right) match {
  case (TypeVariable(x), TypeVariable(y)) => areSubtypes(delta, lookupTypeVar(delta
      ,x), lookupTypeVar(delta,y))
  case (TypeVariable(x), _) => areSubtypes(delta, lookupTypeVar(delta,x), right)
  case (_, TypeVariable(y)) => areSubtypes(delta, left, lookupTypeVar(delta,y))
  case _ => false
}
```

Figure 2.5: nesT Implementation of Subtyping Type Variables

The TRANSS subtyping rule takes into consideration the Type Coercion, as well as creates reflexivity in the subtyping relation as a whole. Because of this, the `areSubtypes()`

method is fully recursive, as shown at the occurrence of a Type Variable as either the left or right arguments of the method call (Fig. 2.5).

Essentially, when a Type Variable appears, a recursive call is made to `areSubtypes()` based on the upper bound of that Type Variable under a given type coercion. This will continue until a subtyping decision is made (either `true` or `false`), which will recursively climb up the tree and return this result. Because of the TRANSS rule of transitivity, the recursively decided result also holds for the original type pair.

```
case Structure(_, leftMemberList) => right match {
  case Structure(_, rightMemberList) => {
    var passedTest = true
    var tempTypeLeft: Representation = Okay
    var tempTypeRight: Representation = Okay
    if (leftMemberList.size < rightMemberList.size)
      passedTest = false
    else {
      for (i <- 0 until rightMemberList.size) {
        val (tempStringLeft, tempTypeLeft) = leftMemberList(i) match {
          case (someString, typeRep) => (someString, typeRep)
          case _ => throw new NesTTypeException(message)
        }
        val (tempStringRight, tempTypeRight) = rightMemberList(i) match {
          case (someString, typeRep) => (someString, typeRep)
          case _ => throw new NesTTypeException(message)
        }
        if ((!(areSubtypes(delta, tempTypeLeft,tempTypeRight))) || (!(
            tempStringLeft==tempStringRight)))
          passedTest = false
      }
    }
    passedTest
  }
  case _ => false
}
```

Figure 2.6: nesT Implementation of Subtyping Structures

Finally, the subtyping rule STRUCTS has been implemented in nesT to provide a relation between nesT Structure types. The STRUCTS rule essentially says that a structure is a subtype of another structure if it shares the same fields. The "supertype" structure can have less fields than the "subtype", as long as each field represented in the "supertype" has a representation in the "subtype" (this is represented in the STRUCTS rule by the operation $\uplus$). Due to the inductive definition of the STRUCTS rule, the nesT implementation of structure subtyping will recursively call the subtyping relation on the types of each structure field that shares the same field identifier. To satisfy the structure subtyping relation, each field in the "supertype" structure must have a representation in the "subtype" structure that has the same identifier and whose field types also follow the subtyping relation. This is represented in the nesT implementation of the structure subtyping case (Fig. 2.6).

This implementation follows an "innocent until proven guilty" mentality, which is to say that the left and right structures are considered subtypes until a field is found that violates the subtyping rule, under which case the `passedTest` variable will be set to false. The nesT implementation of the rule places a further restriction on the subtyping rule that requires the fields of the subtype to appear in the same order as the fields of the super-type. This is a restriction added to provide uniformity across subtyping of structures such to avoid false rejections of structure subtypes.

2.1.6 nesT Type Variable Promotion

Definition 2.1.1 *The relation* $\ll$ promotes *a type variable:*

$$\frac{T \vdash T(t) \ll \tau}{T \vdash t \ll \tau} \qquad\qquad \frac{\neg\exists t.\tau = t}{T \vdash \tau \ll \tau}$$

In some type checking cases, it may be more important to know the overall structure of a type, as opposed to knowing the type's actual upper bound. For example, in the INDEXT nesT Type Checking case (Fig. 2.8) that type checks Array accesses, it is important to know that the structure being accessed is an array of the form `Array[t]`, though the actual type `t` of the array is not important. In this case, it becomes beneficial to "promote" any Type

Variables that may appear in the code to a structured type, so that it can be confirmed that a Type Variable used in a spot like array access meets the necessary requirements of the type rule.

```
def promote(delta: Map[String, Representation],
            t: Option[Representation]): Representation = {
  val tType = t match {
    case Some(TypeVariable(tvar)) => {
      val tvarType = delta.get(tvar) match {
        case None => throw new NesTTypeException(mesg)
        case newType => newType
      }
      promote(delta, tvarType)
    }
    case Some(structuredType) => structuredType
    case None => throw new NesTTypeException(mesg)
  }
  tType
}
```

Figure 2.7: nesT Implementation of Promotion

In DnesT, the promote operation ($\ll$) bases its judgments off a Type Coercion, and attempts to promote a Type Variable into a concrete type. In nesT, this operation is completed via a method called `promote()` which takes as arguments both the Type Variable symbol table and the Type Variable that it intends to promote. This `promote()` method essentially searches the Type Variable symbol table for an instance of the Type Variable. If it does not exist within the symbol table, the Type Variable is undeclared, which causes a type error. If it does exist within the symbol table, its upper bound is returned as the "promoted" type. In some cases, the upper bound of a Type Variable may be another Type Variable, and for this reason, the `promote()` method is recursive and will continue to call itself until a structured (aka non-Type Variable) result is produced.

Whether a relation is subject to promotion or simple examination via a subtyping relation varies from type rule to type rule. In general, a Type Variable will be examined for a subtyping relation if knowledge of the concrete upper bound type is necessary (such as an integer type in addition), whereas a type has to be promoted if the structure of the type is more important than the type itself. In combination, these two operations provide a powerful source to allow Type Variables to be freely used within the nesT code, contributing to the expressive power of the language.

2.1.7 The Type Checking Algorithm

The core of the nesT type checker is the *nesT type checking algorithm*, a recursive pattern match method that combs through the AST, applying type rules and assigning types wherever necessary in the code. The algorithm works by starting at the root node, crawling down to the each leaf node in the tree, and assigning a type to each node as the algorithm recursively ascends back up towards the root node. These types are assigned based on a type rule that is executed on each node. These nesT type rules are implemented from the DnesT typing rules (Fig. 2.8), with some minor changes and additional type rules being added to create a fully typable environment.

$$\textsc{CastT}\quad \frac{G, T \vdash e : \tau \qquad T \vdash \mathit{compatible}(\tau, \varsigma)}{G, T \vdash (\varsigma) e : \varsigma} \qquad\qquad \textsc{CallT}\quad \frac{G, T \vdash \mathsf{f} : \varsigma \qquad T \vdash \varsigma \ll (\tau)}{G, T \vdash \mathsf{f}() : \tau}$$

$$\textsc{AssignT}\quad \frac{G, T \vdash e_1 : \varsigma_1 \qquad G, T \vdash e_2 : \varsigma_2 \qquad T \vdash \varsigma_2 \preccurlyeq \varsigma_1}{G, T \vdash e_1 = e_2 : \varsigma_1} \qquad\qquad \textsc{StarT}\quad \frac{G, T \vdash e : \varsigma \qquad T \vdash \varsigma \ll \tau\star}{G, T \vdash \star e : \tau}$$

$$\textsc{NameT}\quad \frac{G(id) = \tau}{G, T \vdash id : \tau} \qquad\qquad \textsc{IndexT}\quad \frac{G, T \vdash e_1 : \varsigma_1 \qquad G, T \vdash e_2 : \varsigma_2 \qquad T \vdash \varsigma_1 \ll \tau[] \qquad T \vdash \varsigma_2 \preccurlyeq \mathbf{uint}}{G, T \vdash e_1[e_2] : \tau}$$

Figure 2.8: Typing Rules for Selected Expressions

To begin execution of the algorithm, the pattern matching method is called on the root node of the AST. The method then matches the AST Node's token type (the root will likely be a FILE node in conventional nesT code) and executes a specific set of instructions (the nesT type rule) to assign a type to that node. However, most nesT type rules require knowledge of the type of a node's children to function, so the first step at a new node is always to recursively call the type checking method on each of the node's children. This creates a recursive call tree which will exhaustively crawl through the AST until leaf nodes are found that have concrete types which can be passed up through the tree. This allows any parent nodes to complete their type analysis which may have required information about its children's types. Once a node has full information about its children's types, it can execute its specific type rule and assign itself a type. If it any point a type error occurs during the type rule analysis, the type checker will terminate and give an error pointing to what type of error occurred and what caused the error (including what the expected value was and what value was received). Successful completion of the type checking algorithm will assign a type to the root node (usually a *nesT Module Type* - see Sect. 3.1) and the code will be considered type correct.

Because the nesT type checking algorithm exhaustively searches through every node of the AST, many linking nodes that exist merely to structure the AST will be examined in the type checker. An example of one of these linking nodes is the POSTFIX_EXPRESSION node seen in Fig. 2.2, which merely structures the AST by indicating the presence of a new expression within the operation. Often, these nodes will simply send all of their children to the type checker and assign itself the type of `Okay`, signifying that all of its children type checked correctly. Alternatively, other linking nodes may be assigned the same type as its child, as a means of recursively passing the child's type up the tree. Though the type rules for these linking nodes are vital to the function of the algorithm, the real power of the algorithm comes from the type rules that are examined at expression and statement nodes. Each of these nodes has a type rule that will assign a type to the node if and only if a series of conditions are met that deem that expression (or statement) to be type safe. In the following sections we examine a few of these DnesT typing rules, and describe the process of implementing each rule as part of the nesT type checking algorithm.

2.1.8 Case Analysis: Array Access

```
case ASTNode(NesTLexer.ARRAY_ELEMENT_SELECTION, _, children, parent, _) => {
  val siblingType = getSiblingType(parent, depth)
  val arrayType = promote(typeVars, siblingType) match {
    case Array(aType, _) => aType
    case _ => throw new TypeException(mesg)
  }
  val childType = checkNesTExpression(children(0), depth + 1) match {
    case Some(childType) => childType
    case _ => throw new TypeException(mesg)
  }
  if (areSubtypes(typeVars, childType,Int16)) {
    Some(arrayType)
  }
   else if (areSubtypes(typeVars, childType,UInt16)) {
    Some(arrayType)
  }
  else throw new TypeException(mesg)
}
```

Figure 2.9: ARRAY_ELEMENT_SELECTION (IndexT) nesT Type Checking Case

In nesT, array access is handled at the appearance of a ARRAY_ELEMENT_SELECTION node within the AST. The ARRAY_ELEMENT_SELECTION match case implements the DnesT rule INDEXT (from Fig. 2.8). The INDEXT rule has conditions on two expressions e_1 and e_2, where e_1 has type ς_1 and e_2 has type ς_2. In nesT, e_1 is a sibling node of the ARRAY_ELEMENT_SELECTION, thus ς_1 is found through a call to the `getSiblingType()` method which recursively runs the type checker on a sibling node. Additionally, the index of the array access (aka e_2) will be a child node of the ARRAY_ELEMENT_SELECTION, so the child type (ς_2) is found recursively by calling the type checker on the child node of the

ARRAY_ELEMENT_SELECTION. The rule goes on to say that ς_1 (when promoted) must match the structure of an array of type τ. In nesT, the sibling type is promoted (through the `promote()` method) and matched against to a Type Representation of an array of some type. If the sibling type is an array, the type of that array is stored in the variable `arrayType`. The final condition of the INDEXT type rule is that the access value must be a subtype of `UInt`. This is generalized in nesT by checking whether the access value (the type of the child node) is a subtype of either `UInt16` or `Int16` through a call to `areSubtypes()`. Assuming all of these conditions are met, the result of the type rule will be type τ (aka `arrayType`), and the typing of the ARRAY_ELEMENT_SELECTION node is complete.

2.1.9 Case Analysis: Function Calls

```
case ASTNode(NesTLexer.ARGUMENT_LIST, _, children, parent, _) => {
  val siblingType = getSiblingType(parent, depth)
  val (returnType, parameterList) = promote(typeVars, siblingType) match {
    case Function(rt, pl) => (rt, pl)
    case _ => throw new TypeException(mesg)
  }
  for (i <- 0 until children.length) {
    val childType = checkNesTExpression(children(i), depth + 1) match {
      case Some(cType) => cType
      case _ => throw new TypeException(mesg)
    }
    if (!(areSubtypes(typeVars, childType, parameterList(i))))
      throw new TypeException(mesg)
  }
  Some(returnType)
}
```

Figure 2.10: ARGUMENT_LIST (CallT) nesT Type Checking Case

In some cases, the nesT matching case expands on the DnesT type rule due to the increased expressiveness of the nesT language. One example of this is the implementation of the CALLT rule. In nesT, function calls are handled by the ARGUMENT_LIST node, which corresponds to the argument (or parameter) list of a function call. Much like the

INDEXT case for arrays, the CALLT type rule requires a function `f` of type ς, where ς can be promoted to match a function with a return type of τ. In the nesT ARGUMENT_LIST case, a call is made to the `getSiblingType()` method which returns the type of the sibling of the argument list. This type is then promoted via `promote()` and matched to see whether or not it can resolve to a function with a return type and a parameter list. In DnesT, functions do not have explicit parameters, so in the CALLT rule the work is done and the final type resolves to the return type of the function. The latter is true in nesT, as the function's return type is assigned as the type of the node, but not before checking each parameter sent to the argument list and seeing whether or not it is a subtype of the expected parameter of sibling's function type. It is only after confirming that each supplied parameter's type matches the expected parameter type of the function that the return type is assigned to the node and the match case is completed.

2.1.10 Case Analysis: Variable Resolution

```
case ASTNode(nesTLexer.RAW_IDENTIFIER, ident, _, _, _) => {
  Some(Symbols.lookupVariable(node, ident))
}
```

Figure 2.11: RAW_IDENTIFIER (NameT) nesT Type Checking Case

One important thing to note is that the type checking algorithm relies heavily on the symbol tables (Sect. 2.2.3) that exist within each AST Node. In the DnesT typing rules, information about the type of a given variable is contained in a type environment G. This type environment is similar to a Type Coercion T, but is relevant to value variables and types, as opposed to Type Variables and their upper bounds. A nesT symbol table is the implemented version of one of these type environments and plays a vital part in the type checking system. Looking up a variable's type information is handled by the DnesT rule NAMET, which in nesT is implemented in the RAW_IDENTIFER match case. Any time an identifier appears in the code, say as one of the arguments of an ADDITION node, the node that is found as a child of the statement or expression will be a RAW_IDENTIFER. Like its inspiration, NAMET, the RAW_IDENTIFIER case is very simple in that it simply looks up the type information stored within the symbol table about the given identifier. This is accomplished through a call to the method `lookupVariable()`, which takes as parameters both the node (which contains the symbol table information) and the identifier that it wishes to know about. Assuming the identifier is used in correct scope, a type is returned from the symbol table, and this type is passed up the AST and back to the ADDITION node that used it, which is now able to render a decision based on its own type rule.

The nesT type checking algorithm is simple in design, yet holds a great deal of power through application of the various type rules. Upon completion of the algorithm, the code has not only successfully passed type analysis (giving confidence in type safety), but also will have a final type assigned to the root node. This is crucial to the operation of the Scalaness typer (described in Chapter 3), as the module type (Sect. 3.1) calculated by the nesT type checker will be assigned to a Scalaness level nesT component, and used in the first stage type analysis.

2.2 Additional Support for nesT Type Checking

The core of the type checking system for nesT lies within the type checking algorithm and its application of the nesT type rules. However, there are additional nesT features which support the nesT type system as a whole. Specifically, the infrastructure created to handle declarations and the types of variables, as well as a way to elegantly deal with "Array out of Bounds" type errors. In this section we explain how declaration handling was implemented, along with a novel way of inserting run time array bounds checks into the generated nesC code.

2.2.1 Declarations

As shown in section Sect. 2.1.10, an integral part of the type checking system is the ability to assign a type to any variable that occurs within the code. This is done through a look up to a nesT level *symbol table*, an object that maps string identifiers to nesT level type representations. With a symbol table assigned to each AST Node and a comprehensive algorithm designed to look up any symbol that may exist within the scope of a given node, the types of any variables that occur freely within nesT code can be resolved during execution of the nesT type checker. Of course, for this to function properly, the symbol tables must be comprehensively filled with all identifier and type pairs that exist within a given scope. The insertion of each symbol into its corresponding table occurs during a process known as *AST decoration*.

AST decoration occurs after the nesT code is parsed into an AST, but before the running of the type checker. Decoration of the AST occurs in two major steps: the parsing of all DECLARATION nodes that appear in the code into (*identifier*, `Representation`) pairs, and the insertion of these pairs into the symbol table of a proper AST node. With each AST node assigned a fully decorated symbol table, the nesT type checker can commence with full type knowledge of any variables that may occur anywhere within the code.

2.2.2 The Declaration Parser

The first step in populating the symbol tables is the parsing of each DECLARATION node that appears within the AST into an (*identifier*, `Representation`) pair that can be returned to the symbol table. The AST structure of any DECLARATION node contains all of the information needed to create such a pair, such as the declared type of the variable, the variable's identifier, and any modifiers that may change the structure of the type (like in function, array, or pointer declarations). A single method, `extractDeclaredNames`(),

```
DECLARATION (30)
  uint8_t (141)
  DECLARATOR_LIST (33)
    DECLARATOR (31)
      IDENTIFIER_PATH (60)
        data (119)
      DECLARATOR_ARRAY_MODIFIER (32)
        POSTFIX_EXPRESSION (113)
          64 (29)
```

Figure 2.12: AST of an array declaration: `uint8_t data`[`64`]

handles the parsing of each individual declaration. It takes as a parameters a nesT AST Node and returns a list of (*identifier*, `Representation`) pairs. In general, each declaration returns only one pair, but as nesT allows multiple declarators to exist within a single declaration, the `extractDeclaredNames`() method has the capacity to handle both cases.

Because the structure of a DECLARATION AST is standard, most of the relevant information needed to create the (*identifier*, `Representation`) pair comes from a simple search through the AST. For example, the identifier is always a child of an IDENTIFIER_PATH node, and thus easily found. The rest of the parsing of the declaration involves finding the declared type and determining whether or not any modifiers occur. If there are no modifiers, such as in a simple integer or character declaration, the pair is found and returned with relative ease. However, if a modifier does exist, the `extractDeclaredNames`() method will take extra steps to compute the full type of the declaration. As shown in Fig. 2.12, an array declaration may contain a DECLARATOR_ARRAY_MODIFIER node with a child node that represents the size of the array. Examination of this particular AST would result in a pair of (*data*, `Array(UInt8,64)`) being returned by the declaration parser. Pointer modifiers would work much in the same way, as the existence of one (or multiple) POINTER_MODIFIER nodes would simply wrap the declared type in the necessary number of `Pointer()`s.

A function declaration would operate in much the same way, but with an extra step if the function is declared with any parameters. Because each parameter is essentially a declaration (an identifier and a type), the `extractDeclaredNames()` is recursively called on each parameter of a function declaration, returning a list of parameter types that contribute to the overall type of the original function. Because of this, `extractDeclaredNames()` has special logic designed to handle PARAMETER nodes in much the same way it would handle a DECLARATION, but with the number of steps reduced (such as finding the identifiers for each parameter, which do not factor in to the overall function type).

2.2.3 Symbol Tables

The second half of the AST decoration operation inserts each (*identifier*, `Representation`) pair (aka the *symbol pair*) into the appropriate symbol table. To do this, a target node (usually the parent of the DECLARATION node) is selected. Next a map is created using the symbol pair which maps the identifier to the associated type. Then, the newly created symbol map is assigned as the target node's symbol table, completing the process. If the target node already has an existing symbol table (possible if the target node has multiple DECLARATION node children), the symbol maps are checked for conflicts and merged if none arise. In essence, this completes the AST decoration, as an exhaustive search through the AST for DECLARATION nodes along with the creation and insertion of a symbol pair into the symbol table for each node will leave no declared variable without an associated type. However, the insertion of the symbol pair into the symbol table of only a single node creates an issue, as that symbol information should be accessible to every node within the scope of the target node. This is handled by calls to the symbol table using the `lookupVariable()` method.

When `lookupVariable()` method is called by the type checker (Sect. 2.1.10) using an identifier and a node as parameters, the first thing it will do is check the associated symbol table of the node for an instance of the provided identifier. If the identifier is not mapped in that symbol table, `lookupVariable()` will be recursively called on the parent node, examining that symbol table for a mapping of the identifier. This will continue until either the top of the AST is reached, or the identifier mapping is found. Because nesT scoping operates in a linear way, the lack of the symbol pair in any parent nodes of the original node will imply that the variable is undeclared in the original node's scope, leading to an error. If at any point the symbol pair is found, the corresponding type is returned so that the type checking process can continue.

Executing the AST decoration before running the type checker is an important step, as each symbol pair that is added to the symbol table will contribute vital information to the type checker. However, executing the two processes (AST decoration and type checking) separately can create some issues if a variable name is used more than once in the code. Using the same variable identifier in separate scopes is perfectly legal in nesT, and the symbol tables are designed to keep the two variables separate. However, if a variable is declared in a scope with which it's identifier already has a corresponding symbol pair, this can create an issue. Consider the following example:

```
int16_t x[64] = myArray;
int16_t x = x[5];
```

In this example, we are declaring an array of 16 bit integers called `x` which is being instantiated with a pre-existing array. However, in the next step we re-declare `x` to be simply an `Int16`, not an array, and instantiate it with a member of the array `x`. In some langauges, this may be legal, but due to the fact that AST decoration happens before type checking in nesT, the nesT type checker would only see the more recent declaration of `x` at that scope, and not allow the `x[5]` array access to happen. Because of issues like this that arise with varible shadowing in nesT, we place a restriction on the language that allows variable re-declaration only within different scopes, never within the same one.

In addition to the common variable symbol table, there are two more types of symbol tables that exist within the Symbols object. Each is nearly identical in structure to the variable symbol table, but instead apply to Type Variables and Structure Variables. Because there can be a defined nesT Structure type `X` that is different from a defined Type Variable `X`, and even from a simple variable like an integer `x`, each of these three types of variables has its own symbol table. It is up to the type checker on a case by case scenario to decide which of these three symbol tables to query.

2.2.4 Array Bounds Checking

One of the more common errors that programmers run into when compiling and executing code is an "Array out of Bounds" error. This occurs when the code tries to access an element of an array at an address that is larger than the size of the array itself. In general, this will cause a program to crash and is therefore an error that can be very destructive to the code. However, unlike type errors that can be flagged at compile time, array bounds errors can arise during the run time of a program. This is due to the fact that both the size of an array and the index value of the array access can be undecidable at compile time, as they may be calculated via a user-input or values that are dynamically generated during the execution of the program.

Despite the fact that the type checker cannot catch array bounds errors during compilation, nesT still has a way of elegantly dealing with these types of errors. This is achieved by inserting additional code into the AST during compilation (after the parser and type checker have run) that adds in checks for array bounds errors. If an error is found, an elegant user defined function will trigger, usually doing something like rebooting the node, or

Before Specialization:

```
for( i = 0; i < 64; ++i ) {
    message.data[i] = data[i];
}
```

After Specialization:

```
for( i  = 0 ; i  < 64 ; ++i  ) {
  {
    int _sc_1   = i ;
    if( _sc_1  >= 64  )
        call boundsCheckFailed (  );
    {
      int _sc_2   = i ;
      if( _sc_2  >= _sc_data_SIZE  )
          call boundsCheckFailed (  );
      message .data [_sc_1 ] = data [_sc_2 ];
    }
  }
}
```

Figure 2.13: Two array accesses before and after specialization.

re-running the code in an attempt to avoid a full-on crash.

The insertion of these array bounds checks is handled by an object called the "tree transformer". The tree transformer calls a method (`addArrayBoundsChecks()`) with the root node of the full AST as a parameter. The `addArrayBoundsChecks()` method then combs the entire AST looking for instances of array accesses, and upon discovery of one of these accesses, inserts an array bounds check directly into the existing AST. The core AST is modified (through rearrangement and insertion of new nodes) to open a new scope, declare temporary variables to correspond to access values, and create a conditional statement which allows the array access to occur if and only if the access value of the array is within bounds of the size of the array. Failure of the condition results in the execution of a user defined method called `boundsCheckFailed()` (which may reboot the node or perform a similar fail-safe task). Because all of the new code that is inserted into the original AST is written in a standardized type safe way, none of these AST modifications will violate the type result that was achieved during the type checking at compile time of the code. This is a result of the bounds checks being added within new compound statements that open up a new scope and maintain the flow and direction of the code, as seen in the second half of Fig. 2.13. At the conclusion of the tree transformer, the AST has been modified in a way that adds type safe nesT-style AST nodes to already existing type correct code. The result of this is eventually having the specialized code (with added array bounds checks) written into a nesC file that can be safely executed at the mote level.

Chapter 3

Scalaness Typechecking

This chapter will focus on the first stage language Scalaness, which is an implemented version of the theoretical language DScalaness (Chapin, Skalka, Smith, and Watson 2013). The Scalaness language is implemented by extending, through direct compiler modification, the object oriented language Scala to incorporate the full syntax of DScalaness (presented in Fig. 3.1). The DScalaness syntax itself extends the object-oriented core calculus found in Featherweight Generic Java and Featherweight Assignment Java (Igarashi, Pierce, and Wadler 2001; Molhave and Petersen 2005) to allow the manipulation of DnesT Modules. A full specification for the transformation of the DScalaness syntax into Scalaness is given in Sect. 4.2. The major new addition to Scalaness is support that was introduced for the manipulation of nesT Modules, which are objects that contain components of nesT code that can be specialized by the first stage program. The result of the execution of a Scalaness program are files of sensor-network ready *nesC* code, which are generated from these components of nesT code that are type checked and specialized during the Scalaness compilation. Because of the introduction of these new nesT Modules, the Scala type checker has been modified and extended to incorporate a series of type rules designed to verify the type correctness of the three main operations involving nesT Modules (instantiation,

$$
\begin{array}{lcll}
\mathtt{L} & ::= & \mathtt{class}\ \mathtt{C}\langle\bar{\mathtt{X}} <: \bar{\mathtt{N}}\rangle\ \mathtt{extends}\ \mathtt{N}\ \{\bar{\mathtt{T}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} & \textit{classes} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{T}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} & \textit{constructors} \\
\mathtt{M} & ::= & \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} & \textit{methods} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}\langle\bar{\mathtt{T}}\rangle(\bar{\mathtt{e}}) \mid (\mathtt{N})\mathtt{e} \mid & \textit{expressions} \\
 & & \mathtt{l} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{def}\ \mathtt{x} : \mathtt{T} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid & \\
 & & \mathtt{abbrvt}\ \mathtt{X}(\bar{\mathtt{X}}) = \mathtt{T}\ \mathtt{in}\ \mathtt{e} \mid & \\
 & & \mu \mid \mathtt{e} \ltimes \mathtt{e} \mid \mathtt{e}\langle\bar{\mathtt{e}};\bar{\mathtt{e}}\rangle \mid \mathtt{image}\ \mathtt{e} & \\
\mathtt{T} & ::= & \mathtt{X} \mid \mathtt{N} \mid T \circ \mu_T & \textit{types} \\
\mathtt{N} & ::= & \mathtt{C}\langle\bar{\mathtt{T}}\rangle & \textit{class types} \\
\mathtt{l} & ::= & (\mathtt{p}, \mathtt{N}) & \textit{references}
\end{array}
$$

Figure 3.1: The Syntax of DScalaness

composition, and imaging).

With the exception of these new type rules, the type checking system of Scalaness is identical to that of Scala, and therefore these new type rules have been implemented as additional rules that are only to be checked in the presence of a nesT Module Type, leaving the rest of Scala's type theory untouched and therefore maintains Scala's type correctness (Cremet, Garillot, Lenglet, and Odersky 2006). This makes Scalaness a *conservative extension* of Scala, as the new rules added are *more* strict, and will never allow anything that Scala would reject. For this to work, additional information is supplied to the type checker when a nesT Module Type appears, and this information is used to decide whether or not a given operation is allowed under the Scalaness type rules (outlined in Sect. 3.3). If an operation is deemed type safe by Scalaness' additional type rules, then the work of the Scala type checker will continue as normal, whereas if a Scalaness type error is found, it will be treated much like a Scala type error and interrupt compilation. Due to the principle of cross-stage type safety that motivates DScalaness, any code that passes Scalaness type checking is expected to generate type safe code. This means that any nesC that is generated as a result of the specialization of the nesT code is considered to be type correct without the need for any additional type checking before the code is deployed for actual use in a sensor network.

3.1 nesT Module Types in Scalaness

The foundation of the Scalaness language is built around the manipulation of modules of nesT level code that are specialized during the run time of the first stage program. These modules are treated as first class values and have their own strict type discipline. Thus, we introduces a new nesT level type called a *nesT Module Type* that corresponds to a nesT Module:

$$\mu\tau = \langle T; V\rangle\{\iota; \varepsilon\}$$

$\mu\tau$ is the DnesT representation of a module type, which has an equivalent nesT Type Representation:

```
Module(typeParameters, valueParameters, imports, exports)
```

These nesT Module Types contain lists of both type and value parameters that appear in the code, as well as any functional imports or exports that may be required by the module. Depending on the state of the module (un-instantiated, instantiated, runnable, wired, etc.), there may be values contained in some or all of these fields. Each field of the module type is crucial in the typing of any module operations, as shown by the type rules in Sect. 3.3. To be used in Scalaness type rules, these nesT level module types are transformed into Scalaness types by the MODT type rule (Sect. 3.3.1).

Because this nesT Module Type is a new type that is unknown to Scala, it must be introduced to the Scalaness type checker in a way that doesn't interfere with Scala's conventional type checking. Because nesT Modules are treated as first class values, they are usually associated with a Class or Object type in Scala. Replacing this type with a brand new type would cause confusion within the Scala type checker, and result in a necessary rewriting of nearly every Scala type rule. To avoid this, a value's nesT Module Type is included as extra information stored within a conventional Scala type, rather than as the type itself.

In the Scala type checker, each value is assigned a type, all of which are subsets of the blanket `Type` object in Scala. Manipulation of a `Type` object is done during Scala type checking, and is a compiler-only process. Because every type is a subset of this global `Types` object, a new field was added in to `Types` called `nesTModuleType`. This field is set up as a Scala `Option`, meaning the field can either be `Some(TypeRepresentation)` or `None`. So for every type that does not correspond to a nesT Module, the field is simply left empty. This allows the Scala type checker to function as normal with minimal interference, while also allowing additional checks to be made any time a nesT Module Type appears in the code or is required by an operation. These additional checks are described in Sect. 3.2.

Another addition to the Scalaness language that comes into play in the use of nesT Module Types are `LiftableTypes` and `MetaTypes`. Because type information (specifically type and value parameters) cross the barrier between the first and second stage languages, each with its own type set, there must be some concept of serialization between the two. In Scalaness, values and types that are used to instantiate a nesT Module have to be "lifted" (or serialized) into the nesT language where they will become nesT values with nesT Type Representations. The field of types that can satisfy this property are known as `LiftableTypes`. This allows manipulation of Scala level values that correspond to nesT level types within the Scalaness code. These nesT level types can be calculated dynamically during run time of the Scalaness program, thus introducing the need for `MetaTypes`. In Scalaness, a `MetaType` essentially corresponds to a nesT level Type Variable, and is bounded by a `LiftableType`. An un-instantiated nesT Module at the Scalaness level will have a set of type parameters which are nesT level Type Variables bounded by some nesT

Type Representation. When the nesT Module is instantiated, the Type Parameter will be resolved by a dynamically generated value stored in a `MetaType`. Assuming the upper bound of the `MetaType` (when lifted) is a subtype of the upper bound of the expected nesT Type Variable, the instantiated type parameter will be allowed. This is explained more by the INSTT type checking rule in Sect. 3.3.2.

3.1.1 Scalaness Type Annotations

The addition of nesT Module Types is integral in the function of the Scalaness typer, especially due to all of the information contained within a nesT Module Type. However, because Scala has no support for nesT Modules, the type of any nesT Module operation would be undecidable without hacking the Scala compiler to add this new component to the language. Instead, we rely on existing Scala syntax and type forms to introduce decidability to module types via the introduction of *Scalaness Type Annotations*. Scalaness Type Annotations are required on any aspect of the Scalaness code that concerns a nesT Module. This includes class and value definitions that store nesT Modules, module operations (such as instantiation and wiring) that return nesT Modules, and any method parameters that expect a nesT Module. By annotating these constructs with a full module type, the Scalaness compiler can perform its necessary type checking on the code to determine whether or not the manipulation of the module is being done appropriately or in a type correct way.

Scalaness Type Annotations take the form of an extended Scala annotation known as `ModuleType()`. Like any Scala annotation, the argument of `ModuleType()` is a string, which in this case contains a long form written version of all of the nesT type information contained within the module type. This includes Type Variables known to the module, type and value parameters of the module, as well as the type information about any functional imports and exports. A sample annotation for the result of a module wiring is shown in Fig. 3.2.

```
@ModuleType("""{ checksumType <: UInt32 } <;>
               { startPeriodic(period: UInt32): Void;
                 booted(): Void,
                 fired(): Void  }""")
val wireModule = formattingModule +> checkingModule
```

Figure 3.2: Module Composition with a Scalaness Module Type Annotation

Because Scala annotations are visible to the compiler, this information is available during the type checking stage. Though the annotation itself contains just a simple Scala level String object, the Scalaness compiler can use this information to assign the operation an actual nesT Module Type Representation. Each time an annotation appears, the string is parsed into a nesT Type Representation of a module type using a special *Annotation to Type* parser. Assuming the string literal that represents the type was written correctly according to the specifications of the Annotation to Type parser, the result is a nesT Module Type Representation. This annotated type contains all of the type information required to successfully type the operation that it annotates (explained further in Sect. 3.2).

Though there are Scalaness operations which may not require all of the information held within one of these Scalanes Type Annotations, Scalaness requires the use of a full annotation on every declaration or operation that returns a nesT Module Type. This creates decidability in the Scalaness type system at only a small amount of additional work by the programmer. This additional work is made even more simple with the addition of Type Abbreviations that can be used in Scalaness Type Annotations (see Sect. 4.1).

3.1.2 Storing Type Information in Singleton Types

The addition of the `nesTModuleType` field in all Scalaness types allows any conventional Scala type to store additional nesT Module Type information. Specifically, we can store module type information within the type that Scala assigns to an object or value that represents a nesT Module. However, due to the complex nature of the Scala type checker, the assigned Scala type of a single value can appear in multiple forms at different stages of type checking. To combat this, Scalaness will always store nesT Module Type information within a *singleton type* that is associated with a symbol that represents the nesT Module. This avoids conflicts that may arise when the default type that Scala assigns to a value is different than what is expected by Scalaness. For example, suppose a new class is defined in Scalaness that represents an un-instantiated nesT Module:

```
class RadioC extends NesTComponent { ... }
```

Next, a new value is declared to be of the class RadioC:

```
val myRadio = new RadioC
```

Finally, `myRadio` is instantiated and this instantiated module is assigned to a value:

```
val instRadio = myRadio.instantiate( ... )
```

The default Scala type for both `myRadio` and `instRadio` is `class RadioC`, which represents an un-instantiated module type. This would be the correct module type for `myRadio`, but not for `instRadio`. Thus, we store each individual module type in the singleton types that correspond to each symbol (`myRadio.type` and `instRadio.type`). Now if module type information is ever required for either symbol, we know that the module type stored within the singleton type will necessarily be the correct one. One additional requirement this system places on the user is the need to "let-expand" the code by assigning the result of any module operation to a new value so that its symbol's singleton type may be used to store type information.

The use of only singleton types to store the nesT Module Type information is different than the way Scala conventionally does its type checking, but these types are never returned or substituted for the types that the Scala typer uses. Therefore, looking up and storing information in these singleton types will never alter anything about the various types that Scala assigns to the nesT Modules, and Scala's type checking can continued unimpeded. Thus, the only requirement for these singleton types is to accurately represent information about a given nesT Module Type anywhere that one of these nesT Modules may appear. Additionally, because of the syntactic requirement placed on all of these nesT Modules requiring them to be bound to a symbol, the singleton type for that symbol will always be accessible wherever a nesT Module is used. These singleton types use the same scoping as the symbols themselves in Scala, meaning that nesT Module Type information will never be presented out of scope, nor missing when it is expected. This is a unique approach to the problem of adding nesT Module Types to Scalaness, and because the use of nesT Modules is only allowed in certain operations and definitions, we can be confident that using singleton types is appropriate in each situation where a nesT Module can appear.

3.2 The Scalaness Type Checking Algorithm

Much like the nesT type checking algorithm (Sect. 2.1.7) does for nesT, the *Scalaness type checking algorithm* represents the core of the Scala type checking system. However, unlike nesT, the Scalaness type checking algorithm is not built from the ground up, but rather implemented as an extension Scala's already existing typer. As described in the introduction to this chapter, extending the Scala compiler to only add additional rules and checks (while never allowing anything that Scala would have disallowed) maintains the well founded type result of the Scala type checking system. The additional checks (aka the Scalaness type rules - Sect. 3.3) that are added as part of the Scalaness type checking algorithm will only reject type errors that Scala would not have known about, thus constructing a stricter type discipline.

The original Scala typer works in a way that is very similar to the nesT type checking algorithm, in that it crawls through the AST, matching Scala `Tree` types to specific cases that execute rules based on the type of `Tree` that is found. Though there are dozens of these cases, the new Scalaness checks are required in only four: Class Definitions, Object Defintions, Value Definitions, and functional Applys. This is because these are the only cases within the Scalaness type checker that a nesT Module Type could possibly appear. Each of the original Scala cases that handle these `Trees` have been extended to include additional actions. Otherwise, the Scala typer operates as normal and analyzes the Scalaness code as if it were regular Scala.

The new Scalaness code that is added to the `typedClassDef`, `typedModuleDef` (Objects), and `typedValDef` cases operate very similarly. Because Scalaness requires each of these definitions to be annotated with a nesT Module Type, the first additional step taken in any of these rules is to look for annotations and, if one exists, parse the annotation string into a nesT Module Type. The next step involves finding the computed nesT Module Type that is associated with the body of the definition. In a value definition, this type simply comes from the right hand side of the definition (usually the result of some sort of module operation). However, in an object or class definition (which defines a new Module type entirely), this computed type comes from the execution of the nesT Type Checker on code provided within the definition. Each class or object definition that is intended to represent a nesT Module in Scalaness includes a reference to a nesT file which contains the relevant nesT code. It is during the execution of the Scalaness typer (when analyzing a class or object definition) that the nesT type checker is actually run on this code, returning a computed nesT Module Type (as described in Sect. 2.1.7). Once the computed nesT Module Type is found, this is compared with the annotated type, and if there is no type mismatch, the nesT Module Type is assigned to the defined symbol (via the process described in Sect. 3.1.2) completing the additional Scalaness checks.

The final case that requires additional Scalaness checks is Scala's `doTypedApply`. This case represents the execution of any functions or operations within the Scala code. Specifically within Scalaness, this is used to execute the type rules (Sect. 3.3) that deal with module instantiation, composition, and imaging. An `Apply` tree in Scala contains information about both the function and the arguments that are being applied to that function. Using this information, the Scalaness typer looks at the name of the function being applied, and if it matches any of Scalaness' module operations (such as *instantiate*), an additional check is performed. This additional check is actually a call to the specific type rule associated with the function that is being applied. These type rules are implemented as methods which take as arguments the module types being operated on and any additional parameters that may be involved as well (such as the type and value parameters in instantiation). The result of each of these type rules is a returned nesT Module Type, which is attached to the type that Scala computes as a result of the operation (via the `nesTModuleType` field). This will eventually find its way to the right side of a value definition (as explained above), where it will be stored in a singleton type.

Through these additional Scalaness type rules (as well as the infrastructure required to parse nesT Module Type Annotations and store these new types within the singleton types of symbols) the Scala typer has been adapted to create a new type checking system that combines two stages of type checking into one type result: guaranteeing safe execution of not only the first stage Scalaness code, but also the generated second stage nesC code run at the mote level.

MODT

$$\frac{\mu : \mu\tau \text{ in DnesT type checking}}{\Gamma \vdash \mu : \varnothing \circ \mu\tau}$$

MODIMAGET

$$\frac{\Gamma \vdash \mathtt{e} : T \circ \texttt{<>}\{\iota;\varepsilon\} \qquad \text{main}() : \tau \in \varepsilon}{\Gamma \vdash \mathtt{image}\ \mathtt{e} : T \circ \texttt{<>}\{\iota;\varepsilon\}}$$

MODINSTT

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \varnothing \circ \texttt{<}\bar{t} \preccurlyeq \bar{\tau}_1; \bar{x} : \bar{\tau}_2\texttt{>}\{\iota;\varepsilon\} \\ \Gamma \vdash \bar{\mathtt{e}}_1 : \mathtt{MetaType}\langle\bar{\mathtt{T}}_1\rangle \qquad \Gamma \vdash \bar{\mathtt{e}}_2 : \bar{\mathtt{T}}_2 \qquad \vdash [\![\bar{\mathtt{T}}_1]\!] \preccurlyeq \bar{\tau}_1 \qquad \vdash [\![\bar{\mathtt{T}}_2]\!] \preccurlyeq \bar{\tau}_2\end{array}}{\Gamma \vdash \mathtt{e}\langle\bar{\mathtt{e}}_1;\bar{\mathtt{e}}_2\rangle : \bar{t} \preccurlyeq [\![\bar{\mathtt{T}}_1]\!] \circ \texttt{<>}\{\iota;\varepsilon\}}$$

MODWIRET

$$\frac{\Gamma \vdash \mathtt{e}_1 : T_1 \circ \texttt{<>}\{\iota_1;\varepsilon_1\} \qquad \Gamma \vdash \mathtt{e}_2 : T_2 \circ \texttt{<>}\{\iota_2;\varepsilon_2\} \qquad \iota = (\iota_1/\text{Dom}(\varepsilon_2))@\iota_2}{\Gamma \vdash \mathtt{e}_1 \ltimes \mathtt{e}_2 : T_1 \curlyvee T_2 \circ \texttt{<>}\{\iota;\varepsilon_1\}}$$

Figure 3.3: DScalaness Module Typing Rules

3.3 Scalaness Type Rules - Specification and Implementation

The core of the Scalaness typer and its difference when compared to the original Scala type checker are the new type rules that deal with nesT Modules. These four rules (representing the typing of a nesT Module in Scalaness, as well as three operations) each take nesT Modules as arguments and give a well defined rule that explains how they will be handled at the Scalaness level, as well as what their result might be. In this section, each rule is presented in its original DScalaness form, described, and accompanied by an explanation of how they were implemented in Scalaness. The code for each implementation is presented in full in Appendix A and line numbers of the form (#) are referenced in each explanation.

3.3.1 Module Typing

The first type rule in Fig. 3.3 is the ModT rule and deals with the representation of a DnesT Level Module in DScalaness. The rule itself defines a DScalaness Module type μ as an empty Type Map (a list of Type Variables known to the module) combined with a DnesT level module type. In the actual language implementation, a Scalaness level module is a Scala pair, where the first item in the pair is a map from Type Variable to type representation (representing Type Variables and their upper bounds), and the second is the nesT Type Representation for a Module. The implementation of the ModT rule (Appendix A) reflects this, as the rule itself is a simple method that takes a nesT level module type as an argument and returns the type in a pair with an empty Type Map (2). Because all of the future type rules in Scalaness require Scalaness level module types, the implementation of the ModT rule, however simple, provides an effective foundation for the rest of the Scalaness typing.

3.3.2 Module Instantiation

The instantiation of a module is perhaps the most important piece of the Scalaness language due to the direct interaction between the first and second stage languages. Module instantiation is the only place where specialization of the nesT code occurs, and therefore the ModInstT type rule for instantiation and its Scalaness implementation are vital to the function of the type checker. In Scalaness, the ModInstT rule is implemented (Appendix A) as a method which takes as parameters a Scalaness module, a list of the type parameter upper bounds, and a list of the types of each value parameter. The goal of this type rule is to ensure that the supplied type parameters and value parameters that are being used to instantiate the module meet the typing requirements of the module itself. To do this, a pattern match first examines the supplied module and isolates the key components of it (i.e. the type map, the defined type and value parameters, and the imports/exports) (5). The

MODINSTT rule expects the module to have an empty type map to begin with, thus the first thing that is checked is the size of the module's type map (`7`). Assuming it is empty, the instantiation of the type will continue.

Next, three lists are extracted from the type and value parameters of the supplied module's type (`15, 24, 33`). These lists are the list of Type Variables from the type parameters ($\bar{t}$), their corresponding upper bounds ($\overline{\tau}_1$), and the types that are mapped to each value parameter ($\overline{\tau}_2$). These types are all nesT Type Representations, as they come from the type information of a Scalaness module, which recalling the MODT type rule is simply a nesT Module Type with an additional type map.

The next step of the type rule involves comparing the two $\overline{\tau}$ lists with the types that were instantiated with. However, this is where the implementation gets more complicated. Within the `doTypedApply` rule where the MODINSTT is being called, the types of the parameters are contained in a list of strings. This data is gathered by default from the Scala type checker, and therefore must be parsed from a list of strings into a series of nesT Type Representations usable by Scalaness. This list of strings contains both the type and value parameters that are being used in the module instantiation, where the type parameters have the Scala type `MetaType[LiftableType]` (a Scalaness `MetaType` with an upper bound that will be resolved during run time), and the value parameters are all Scalaness `LiftableTypes`. Because part of the MODINSTT rule requires parameters of this form, the method that parses the strings into nesT Type Representation first confirms that all Type Parameters are of the form `MetaType[LiftableType]` and all of the value parameters have some `LiftableType`. If this is true, a method called `liftTypeString()` converts the string that contains the `LiftableType` (whether the type of the value parameters or the upper bound of the `MetaType`) into a nesT Type Representation. At the conclusion of this parsing method (called `stripStrings()`), the Scalaness Type Checker will have two lists of Type Representations, one representing the upper bounds of the `MetaTypes` and the other representing the types of each value parameters. It is these lists that are sent as parameters to the ModInstT type rule, rather than the full parameters themselves.

With the set of lists in hand, the rest of the ModInstT rule's implementation follows the rule itself. The list of lifted `MetaType` upper bounds is compared to the list of type parameter upperbounds ($\overline{\tau}_1$) using a method called `seriesSubtype()` (41), which ensures that each item in a list is a subtype of the corresponding item in another list. This same subtyping relation is called on the list of lifted value parameter types in comparison to the list of expected parameter types ($\overline{\tau}_2$). Assuming both subtyping comparisons are approved, the final type of the instantiated module is constructed ala the typing rule. The type map is a new map with the list of Type Variables ($\bar{t}$) from before, only now they are upper bounded by the lifted `MetaType` list (44). The imports and exports of the module remain the same, and the type and value parameter lists of the instantiated module are empty. This constructed type is returned as a result of the method (46), and the instantiation of the type is considered successful.

3.3.3 Module Wiring

Like the ModInstT rule, the DScalaness rule for the composition (or wiring) of modules relies on a number of conditions and operations in order to arrive at a final type. Specifically, within the ModWireT rule are two operations m/S (a map m where any domain elements in m that exist within S are removed), as well as $m_1 \curlyvee m_2$ (which is simply a combination of the two maps m_1 and m_2 with the condition that an element in the domain of m_1 and m_2 must map to the same value in both maps). Both of these operations are used in the ModWireT rule, thus it was neccessary to create equivalent operations in Scalaness. Thus, m/S was implemented as `removeDomain()` and $m_1 \curlyvee m_2$ became `nonExlcusiveTypeMapMerge()`. Both methods were implemented in accordance with the definition of the operation, so that the Scalaness implementation of the ModWireT rule could be a simple and effective interpretation of the rule.

With the addition of the two helper operations, the Scalaness implementation of ModWireT (Appendix A) was relatively straightforward. The rule itself (like the other Scalaness type rules) is designed as a method that takes two Scalaness module types as arguments and returns a Scalaness module type as a result. A pattern match isolates the imports, exports, and type map associated with each of the two argument module types (`5, 14`), and through a series of method calls and Scala operations, the imports, exports, and type map of the wired result is built. First, `removeDomain()` is called on the imports of module one, removing any values in the domain of the exports of module two (`22`). Once the imports of module one have been changed, they are attached through a cons operation (@ in the DScalaness rule) to the imports of module two (`24`). Together these will become the resulting imports for the wired module. Next, the type maps from both modules are merged using the `nonExlcusiveTypeMapMerge()` operation (`23`). This result will be the final type map of the wired module. The final exports (like in the ModWireT type rule) are simply the exports of module one, which requires no operation (`25`). Finally, a Scalaness module type is created from the final imports, exports, and type map (`27`). This module type is returned as the return value of the operation. If at any point during the composition of the final wired type there is an error (like in the `nonExlcusiveTypeMapMerge()` operation), the wire will fail due a type error and the error will be reported. Because of this, any wire operation that returns a full wire type from the Scalaness implemented wiring type rule will be considered to be well-typed.

3.3.4 Module Imaging

The final type rule in DScalaness is the MODIMAGET rule for imaging a nesT module. After the nesT code has been specialized through the instantiation of a module and all required wirings have been completed, the final resulting module will be imaged. Imaging a module is what takes the newly specialized and re-wired code and outputs it into nesC files that will eventually be run on the sensor nodes. For a module to be imaged, it must have no unresolved (or "dangling") imports, as well as an empty list of type and value parameters (a sign that the module has been instantiated). The implementation of the MODIMAGET rule in Scalaness (Appendix A) enforces this through a pattern match to assure that the list of imports (as well as the type and value parameters) are empty (`3`). If this is the case, the resulting type of the operation is the same type that was passed to the method, ala the MODIMAGET type rule.

Chapter 4

Additional Support for Scalaness

The core of this thesis was the design and implementation of two type checkers, one for nesT and one for Scalaness, that would be the practical manifestation of the theoretical DnesT and DScalaness languages. Both typers were implemented and completed in a way that mirrored the novel functions and features put forth in the foundational languages. However, over the course of actually implementing the languages, the final product sometimes strayed from the original theoretical vision. This is due to practical reasons when actually implementing the language, especially in Scala. Because of this, some of the code and ideas from DScalaness and DnesT were implemented in a fully functional way, but with a more verbose or less elegant syntax. A long term goal of continued Scalaness development would be to manipulate the parser itself to allow for a 1 to 1 correspondence between the syntax of Scalaness and DScalaness. Until then, a complete specification from the theoretical DScalaness to the actually implemented Scalaness language is presented so that any code designed or intended to be run in DScalaness can be properly written, tested, and run with the Scalaness compiler. This specification is given in Sect. 4.2.

In addition to modifications to the language itself, the implementation of Scalaness also gave the programmer the additional task of annotating each Scalaness construct that defined or returned a nesT Module Type. Because of this, it was always the intention for Scalaness to have a way of shortening these long-form Scala strings that represent types into a simple variable that can be used multiple times throughout the code, and even manipulated through interchangeable sections of the long form type to allow more freedom and simplicity when annotating. These were implemented as Type Abbreviations, and are detailed in Sect. 4.1.

4.1 Type Abbreviations

As described in Sect. 3.1.1, Scalaness requires any definition or operation that returns a nesT Module to be annotated with its expected nesT Module Type Representation. Though vital to the function of Scalaness, this creates an extra burden on the programmer, as Scalaness Type Annotations can be quite long and require precision when writing the type string. Additionally, many of these types are used multiple times (either exactly or with slight variations) throughout the code, requiring lengthy and redundant annotations. To relieve this burden for the programmer, Scalaness includes Type Abbreviations - variables that can store type string information to be used in place of a long form annotation.

In their simplest form, a Type Abbreviation represents a Scala level string. This string has the same form and function as a string that is used in a Scalaness Type Annotation. The difference is, when this string is used once, in the definition of a Type Abbreviation, it never needs to be written in long form again. This is of course a basic coding principle of any variable, but Type Abbreviations have 2 key features missing from a typical Scala string variable. One, they can be manipulated and parameterized by the user, and two, their value is known at compile time. The latter is the key reason for using Type Abbreviations and not annotating with common string variables, as their value is unknown to the Scalaness typer. Of course, by default, the value of a Type Abbreviation is unknown at compile time, which is why additional framework was implemented in the Scalaness type checker to support

Type Abbreviations. (See Sect. 4.1.1).

Though helpful even as a representation of a single string literal in Scalaness, Type Abbreviation's utility goes even further given the ability to parameterize Type Abbreviations. This means a Type Abbreviation can be declared with parameter variables in the string, with the intention of later "instantiating" these parameters with whichever type is useful at the time. These parameters can be initialized either with type strings (such as initializing a parameter with *"Int8"* or *"Int16"*), or even with other Type Abbreviations. This provides a number of ways to create flexible, expressive Type Abbreviations that can be used in a variety of annotations, saving considerable time and effort on behalf of the user.

For example, in Fig. 4.1 we see a Type Abbreviation `MesgT` declared that represents a nesT structure called `MessageType`. Additionally, we see another Type Abbreviation called `SendT` declared that represents a module type. `SendT` includes a parameter *"MesgType"* within its type that will later be parameterized with a full type. This parameterization is shown in Fig. 4.2 as `SendT` is used to annotate a value definition by parameterizing *"MesgType"* with the full string value of the abbreviation `MesgT`. Subsequent use of the structure type `MessageType` can follow this same style of parameterization, drastically reducing the number of times to full message type has to be written out. And if another module wanted to use the `SendT` abbreviation but parameterize it with a slightly different *"MesgType"*, this would be allowed as well.

With clever use of Type Abbreviations, the burden of annotating each and every Scalaness operation that results in a nesT Module becomes less and less, minimizing time and frustration for the programmer, all while keeping valuable annotation information available to the Scalaness compiler, and allowing it to function as originally intended.

4.1.1 Type Abbreviations - Implementation

As previously stated, the defining feature of a Type Abbreviation in comparison with a typical Scala variable is the ability to determine what the value of said variable is at compile time. By default, a variable's value is not known during compilation, as that value could be defined elsewhere in the code, or even calculated at run time. The implementation of Type Abbreviations is designed to get around this, as knowing the string value associated with the Type Abbreviation is crucial in creating and type checking Scalaness annotations.

At the Scalaness level, a Type Abbreviation is declared with two parameters: a main type string, as well as a list of substrings that act as parameters to the main type. The list of parameter strings can be empty, which indicates the initial string represents a full type, or it can have any number of string parameters which will eventually be replaced with types during the parameterization of the abbreviation. Example Type Abbreviation declarations are shown in Fig. 4.1.

```
val MesgT = new TypeAbbreviation("MessageType{src: addrT, dest: addrT, data: Array[
    UInt8,64]}", List())
val SendT = new TypeAbbreviation("""{} < addrT <: UInt32; >      {radio(message:
    MesgType): ErrorT; send(s: addrT, d: addrT, data: Array[UInt8]): ErrorT }""",
    List("MesgType"))
val ScodeT = new TypeAbbreviation("""{addrT <: UInt32} <;>       {;send(s: addrT, d:
    addrT, data: Array[UInt8]): ErrorT}""",  List())
```

Figure 4.1: Type Abbreviaton Declarations with and without parameters.

When it comes time to use the Type Abbreviation as part of a Scalaness Annotation, the syntax is different than that of a conventional type annotation. A non-parameterized abbreviation appears simply as a `@TypeAbbr` annotation that refers to the name of the abbreviation, while a parameterized abbreviation makes a call to a method `parameterize()` within that annotation. The arguments of the method `parameterize()` are the type strings that are being used to parameterize the main type, whether they be string literals or other Type Abbreviations. Both cases are shown in Fig. 4.2.

```
@TypeAbbr(SendT.parameterize(List(MesgT.getFullType))
val rawSendC = new SendC

@TypeAbbr(ScodeT)
val scode = (rawSendC.instantiate(addrt)) +> rawRadioC.instantiate(addrt)
```

Figure 4.2: Value Defintions Annotated with Type Abbreviations.

Though these Type Abbreviations are written in the Scalaness code that is executed at run time, each Type Abbreviation declaration and call is mirrored at compile time during execution of the Scalaness typer. What this means is that each time a Type Abbreviation is declared in Scalaness, an identical abbreviation is declared in the scope of the compiler, allowing its value to be freely known during the type checker. Though the variable used to declare the Type Abbreviation in Scalaness is technically still unknown at compile time, an identical variable holding identical information is declared within the type checker, making manipulation of the Type Abbreviations possible.

When a new Type Abbreviation is declared in the Scalaness code (and mirrored in the compiler), it is saved in a method very similar to nesT Module Types. The abbreviation is stored within the singleton type that corresponds to the its symbol, in a special field called `TypeAbbreviation`. Therefore, any time that symbol appears in an annotation, the Type Abbreviation that corresponds to that symbol can be pulled from the `TypeAbbreviation` field of the singleton type, and its type string can be found and parsed in the same way as a normal Scalaness annotation (see Sect. 3.1.1). A Type Abbreviation with parameters simply adds the extra step of replacing each substring with the string used to parameterize the full type before it is sent to the module type parser. After the full type is returned from the parser, it is used in the type checker just as conventional annotation type would have been. Thus, the Type Abbreviation has officially satisfied the requirement of replacing a full Scalaness Type Annotation.

4.2 DScalaness to Scalaness Transformation

The following is a specification for the transformation of code from the theoretical DScalaness language into the practical and implemented Scalaness language. This specification includes the definition of a nesT Module in Scalaness, all module-specific operations, as well transformations for basic constructs like value and method definitions. Also included is an explanation of shorthand notation for Annotations in Scalaness, as this notation is used throughout the rest of the transformation specification. Any DScalaness construct that is not included in the specification is considered trivial, as the transformation is isomorphic from DScalaness to Scalaness.

A translation from DScalaness syntax *d* to Scalaness syntax `S` is denoted:

$$\|d\| \Longrightarrow \texttt{S}$$

4.2.1 Notation for Scalaness Annotations

Module Type Annotations are a vital part of Scala's type checking due to the type information contained within each annotation. As explained in Sect. 3.1.1, every construct that returns a nesT Module (whether it be a value/method definition, a module operation like wiring, or a module definition itself) requires a Scalaness Type Annotation. Because of this, the specification from the DScalaness to Scalaness language must also include reference to these annotations wherever they are required. Thus, we introduce a transformation A(`X`) which transforms a given type `X` into a Scalaness annotation whenever `X` has a nesT Module Type (and does nothing otherwise).

We define A(`X`) for `X` = `T`, $\mu\tau$, `X`($\bar{\texttt{T}}$)

```
A(T)      = null
```

Where `T` is not a nesT Module Type

```
A(Δ ∘ μτ)  =  @ModuleType("""{Δ_1 , ... , Δ_n}
                                < T_1 <: τ_1 , ... , T_n <: τ_n ;
                                  V_1 : ν_1 , ... , V_n : ν_n >
                                ι_1 , ... , ι_n ; ε_1 , ... , ε_n)""")
```

Where $\mu\tau$ = <T; V> $\{\iota, \varepsilon\}$ is a nesT Module Type
and $\Delta = \Delta_1 , \ldots , \Delta_n$ are known type variables
and $\mathrm{T} = \mathrm{T}_1 <: \tau_1 , \ldots , \mathrm{T}_n <: \tau_n$ are the type parameters
and $\mathrm{V} = \mathrm{V}_1 : \nu_1 , \ldots , \mathrm{V}_n : \nu_n$ are the value parameters
and $\iota = \iota_1 , \ldots , \iota_n$ are the imports
and $\varepsilon = \varepsilon_1 , \ldots , \varepsilon_n$ are the exports

```
A(X(T̄))      =  @TypeAbbr(X.parameterize("T_1", ... , "T_n"))
```

Where X is a Type Abbreviation that stores a nesT Module Type
and $\bar{\mathrm{T}} = \mathrm{T}_1 , \ldots , \mathrm{T}_n$ are the instantiating parameters

4.2.2 Module Definitions

The following transformation specifies the definition of a new nesT Module Class in Scalaness, including its parameters, imports, exports, and method of instantiation.

$\|$<T; V>{ι; $\overline{d}$; ξ}$\| \Longrightarrow$

```
A(τ_new)
class X extends NesTComponent {
  var sclnsT_1 : MetaType[LiftableType] = null
                     ⋮
  var sclnsT_n : MetaType[LiftableType] = null
  var sclnsV_1 : LiftableType = null
              ⋮
  var sclnsV_n : LiftableType = null
  def instantiate( V_1 : LiftableType , ... , V_n : LiftableType ,
    T_1 : MetaType[LiftableType], ... , T_n : MetaType[LiftableType
      ]) {
      val result = new X
      result.sclnsV_1 = V_1
                ⋮
      result.sclnsV_n = V_n
```

```
        result.sclnsT$_1$ = T$_1$
                 ⋮
        result.sclnsT$_n$ = T$_n$
    }
    "X.nt"
  }
```

Where $\mathrm{T} = \mathrm{T}_1, \ldots, \mathrm{T}_n$ are the type parameters

and $\mathrm{V} = \mathrm{V}_1, \ldots, \mathrm{V}_n$ are the value parameters

and $\tau_{new} = \varnothing \circ$ <T; V> $\{\iota, \varepsilon\}$ is the annotated module type

and ε is the type signature of ξ

4.2.3 nesT Module Operations

The following transformations are on nesT Module operations. They all take nesT Modules as arguments and return nesT Modules as results. For simplification of the type theory, we place a syntactic restriction requiring the arguments of each operation to be an identifier. This can be achieved through "let-expansion" of the code using the **Value Definition** rule in Sect. 4.2.4. Because the result of each operation will be applied to a value definition, Scalaness Type annotations are not included in this transformation.

Module Wiring:

$\|\mathrm{x} \ltimes \mathrm{y}\| \Longrightarrow$

```
x +> y
```

Module Instantiation:

$\|\mathrm{x}$ <$\bar{\mathsf{e}}_T$,$\bar{\mathsf{e}}_V$>$\| \Longrightarrow$

```
x.instantiate(e$_{T1}$, ..., e$_{Tn}$, e$_{V1}$, ..., e$_{Vn}$)
```

Where $\bar{\mathsf{e}}_T = \mathsf{e}_{T1}, \ldots, \mathsf{e}_{Tn}$ are the instantiating type parameters

and $\bar{\mathsf{e}}_V = \mathsf{e}_{V1}, \ldots, \mathsf{e}_{Vn}$ are the instantiating value parameters

Module Imaging:

$\|\mathrm{image(x)}\| \Longrightarrow$

```
x.image()
```

4.2.4 Conventional Constructs

This section specifies the transformation of value and method definitions from DScalaness to Scalaness. Any instance of `T` in the DScalaness syntax signifies type information and is therefore used in the annotation transformation defined in Sect. 4.2.1. As previously defined, any nesT Module Type will result in an annotation, while any non-module type will have no additional information added in Scalaness. Additionally, the presence of any DScalaness syntax d in appearing on the Scalaness syntax as $\|d\|$ refers to a recursive call to the transformation.

Value Definition:

$\|\mathtt{def}\ \mathtt{x} : \mathtt{T} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\| \Longrightarrow$

```
{
   A(T) val x = ||e1||;
   ||e2||
}
```

Method Definition:

$\|\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\| \Longrightarrow$

```
A(T) def m(A(T1) x1 : T1, ..., A(Tn) xn : Tn)
{
  return ||e||;
}
```

Where $\bar{\mathtt{x}} = \mathtt{x}_1, \ldots, \mathtt{x}_n$ are the method parameters
and $\bar{\mathtt{T}} = \mathtt{T}_1, \ldots, \mathtt{T}_n$ are their types

4.2.5 Type Abbreviations

This section gives a specification for the translation of a Type Abbreviation from DScalaness to Scalaness. As described in Sect. 4.1, Scalaness Type Abbreviations are declared with strings that represent the type that is stored within the abbreviation, along with a list of sub-strings that act as parameters. The string that represents the type must eventually resolve to a string that is parsable by the Annotation to Type Parser as described in Sect. 3.1.1.

Type Abbreviation:

$\|\texttt{abbrvt x}(\bar{\texttt{x}}) = \texttt{T in e}\| \Longrightarrow$

```
{
    val x = new TypeAbbreviation("T", List("x_1", ... , "x_n"))
    ||e||
}
```

Where "T" is the string representation of T

and $\bar{\texttt{x}}$ = "$\texttt{x}_1$", ... , "$\texttt{x}_n$" are the substring parameters of "T"

Chapter 5

Conclusion

The goal of this thesis was to provide explanation, justification, and insight into the process of building a two-stage type checking system for the Scalaness and nesT programming languages. The novelty of the two staged approach to language design allows increased freedom and complexity of design for a sensor node that is severely deprived of resources. This thesis has presented an approach to this type checking system that has actually been implemented and tested in a way that produced real results. The system has not only been applied to several small working samples, but has also been used to implement the SpartanRPC network security system (Chapin and Skalka 2010; Chapin and Skalka 2013). The SpartanRPC system implementation in Scalaness/nesT uses public key cryptography throughout a network of sensor nodes by specializing sensor level code with session keys computed during the first stage (Chapin 2014). Through implementation of the foundational type principles of DScalaness/DnesT, the type checking system in Scalaness/nesT has demonstrated confidence in the type safety of second stage code that has been generated from well-typed first stage code, thus satisfying, in theory, the principle of cross-stage type safety.

Appendix A

Scalaness Type Rule Code

The following code implements the DScalaness Typing Rules in Fig. 3.3.

Implementation of the MODT type rule in Scalaness.

```
def toModuleType(mod: Module): (Map[TypeVariable, Representation], Module) = {
  (Map[TypeVariable, Representation](), mod)
}
```

Implementation of the MODINSTT type rule in Scalaness.

```
def typeInstantiate(
    mod: (Map[TypeVariable, Representation], Module),
    listT1: List[Representation], listT2: List[Representation]): (Map[TypeVariable,
        Representation],Module) = {

  val (typeParList, valParList, impList, expList) = mod match {
    case (typeMap, Module(typePars, valPars, imports, exports)) => {
      if (typeMap.size == 0)
        (typePars, valPars, imports, exports)
      else throw new Exception("bare module type required")
    }
    case _ => throw new Exception("bare module type required")
  }

  // Upper bounds on what is asked for in module declaration
  val tList = for (i <- 0 until typeParList.length) yield {
    val currT = typeParList(i) match {
      case (TypeVariable(x), someType) => TypeVariable(x)
      case _ => throw new Exception("expected type variables")
    }
    currT
  }

  // Upper bounds on what is asked for in module declaration
  val tau1List = for (i <- 0 until typeParList.length) yield {
    val currTau = typeParList(i) match {
      case (TypeVariable(x), someType) => someType
      case _ => throw new Exception("expected type variables")
    }
    currTau
  }

  // Upper bounds on what is asked for in module declaration
  val tau2List = for (i <- 0 until valParList.length) yield {
    val currTau = valParList(i) match {
      case (someString, someType) => someType
      case _ => throw new Exception("expected some value")
    }
    currTau
  }

  if (!(seriesSubType(listT1,tau1List.toList) && seriesSubType(listT2, tau2List.
      toList)))
    throw new Exception("type and value parameters require subtype relation")

  val typeVarMap = newTypeMap(tList.toList, listT1)

  (typeVarMap, Module(List(),List(),impList,expList))
}
```

Implementation of the MODWIRET type rule in Scalaness.

```
  def typeWire(
      modOne: (Map[TypeVariable,Representation], Module),
      modTwo: (Map[TypeVariable,Representation], Module)): (Map[TypeVariable,
          Representation], Module) = {
```

```

  val (typeMapOne, typeOne, valOne, impOne, expOne) = modOne match {
    case (typeMap, Module(typePars, valPars, imports, exports)) => {
      (typeMap, typePars, valPars, imports, exports)
    }
    case _ => {
      throw new Exception("require a module type during wiring")
    }
  }

  val (typeMapTwo, typeTwo, valTwo, impTwo, expTwo) = modTwo match {
    case (typeMap, Module(typePars, valPars, imports, exports)) => {
      (typeMap, typePars, valPars, imports, exports)
    }
    case _ => {
      throw new Exception("require a module type during wiring")
    }
  }
  val impOneRemoved = removeDomain(impOne, expTwo)
  val typeMap       = nonExclusiveTypeMapMerge(typeMapOne.toList, typeMapTwo.toList)
  val imports       = impOneRemoved:::impTwo
  val exports       = expOne

  (typeMap.toMap, Module(List(),List(),imports,exports))
}
```

Implementation of the MODIMAGET type rule in Scalaness.

```
def typeImage(mod: (Map[TypeVariable, Representation], Module)): (Map[TypeVariable,
    Representation], Module) = {
  mod match {
    case (typeMap, Module(List(), List(), List(), exports)) => {
      (typeMap, Module(List(), List(), List(), exports))
    }
    case _ => throw new Exception("image expects runnable module type")
  }
}
```

Bibliography

Chapin, P. (2014). *Trust Management in Distributed Resource Constrained Embedded Systems*. Ph. D. thesis, University of Vermont.

Chapin, P. and C. Skalka (2010). Spartanrpc: Wsn middleware for cooperating domains. In *IEEE Conference on Mobile and Ad-Hoc Sensor Systems*.

Chapin, P. and C. Skalka (2013). Spartan RPC. Technical report, University of Vermont. Submitted. `http://www.cs.uvm.edu/~skalka/skalka-pubs/chapin-skalka-spartanrpctr.pdf`.

Chapin, P., C. Skalka, S. Smith, and M. Watson (2013, October). Scalaness/nesT. type specialized staged programming for sensor networks. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences (GPCE '13)*.

Consel, C., L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé (1998). Tempo: specializing systems applications and beyond. *ACM Comput. Surv.*, 19.

Cremet, V., F. Garillot, S. Lenglet, and M. Odersky (2006). A core calculus for scala type checking. In *Proceedings of the 31st international conference on Mathematical Foundations of Computer Science*, MFCS'06, Berlin, Heidelberg, pp. 1–23. Springer-Verlag.

Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler (2003). The nesC language: A holistic approach to networked embedded systems. In *PLDI*.

Ghelli, G. and B. Pierce (1998). Bounded existentials and minimal typing. *Theoretical Computer Science 193*(1-2), 75 – 96.

Igarashi, A., B. C. Pierce, and P. Wadler (2001). Featherweight Java. *ACM Trans. Program. Lang. Syst. 23*(3), 396–450.

Liu, Y., C. Skalka, and S. Smith (2011). Type-specialized staged programming with process separation. *HOSC*, 341–385.

Molhave, T. and L. H. Petersen (2005). Assignment Featherweight Java. Master's thesis, University of Aarhus.

Odersky, M., L. Spoon, and B. Venners (2011). *Programming in Scala, second edition*. Artima, Inc.

Taha, W. (2004). Resource-aware programming. In *ICESS*, pp. 38–43.

Taha, W. and T. Sheard (1997). Multi-stage programming with explicit annotations. In *PEPM*, pp. 203–217.