Newton: A Language for Describing Physics

by

Jonathan Lim

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

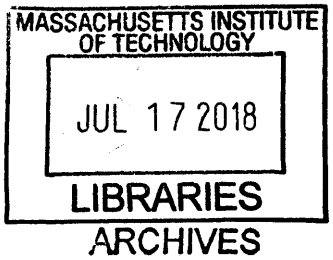
~~June 2017~~ [September 2017]

Signature redacted

Author.....
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by. Signature redacted
Martin Rinard
Professor
Thesis Supervisor

Accepted by.. Signature redacted
Christopher Terman
Chairman, Master of Engineering Thesis Committee



Abstract

Sensors embedded within hardware platforms such as smart-watches and cars read in streams of data. These sensor data may be related to each other by invariants or may have other value constraints, but computing in sensor platforms currently ignores these invariants between sensor data. If the programmer wanted to exploit these invariants to perform safety checks or optimize performance, she has to hard-code the invariants in the program. To exploit invariants in software automatically, each compiler of the language used for every sensor platform could be modified to be aware of different sets of invariants in the programs it compiles, or the compilers could take in a configuration file that describes these invariants. This MEng thesis introduces Newton, a language in which the configuration files can be written, as well as a compile-time library and a runtime library that can be used by other compilers to make compile-time transformations to their source code and exploit the invariants in a Newton description at runtime. We introduce two compile-time algorithms that transform intermediate representations of other compilers. The first transformation adds reliability by checking invariants on program variable values at runtime and by running an error handler function if invariants are violated. The second transformation trades off reliability gained from sensor redundancy for performance by removing code that deals with redundant sensors. This thesis describes twelve examples of realistic physical systems that may benefit from using Newton.

Acknowledgments

First, I want to thank Phillip Stanley-Marbell for giving me guidance throughout this project. He has always been incredibly helpful through all parts of designing Newton and has taught me how to think about research. He is an amazing mentor, and this work would not have been possible without him.

I also want to thank Professor Martin Rinard for being supportive and giving me the opportunity to work on this project.

Last but not least, I want to thank my parents who made my education possible and were always there for me when I needed them.

Contents

Chapter 1

Introduction

Sensor data of embedded physical systems follow certain invariants governed by constraints on hardware or natural laws of physics. The invariants represent the relationships between the values of sensor readings and the constraints on those values on a given hardware platform. A sensor platform is a hardware environment that contains individual sensors made by component suppliers — examples include smart-watches, automobiles, and smartphones (see Figure 1-1). Since different sensor platforms have their own use cases, the constraints that apply to sensor data of those platforms are different as well. For example, accelerometer data of an airplane are constrained by its motion, and hardware specs of automobile engines have maximum and minimum temperatures that they can tolerate. Tangential velocity and angular velocity sensor data of a rotating robotic arm are related by radius, and thermometer readings and barometer readings of a gas in a weather balloon are related to each other by the ideal gas law.

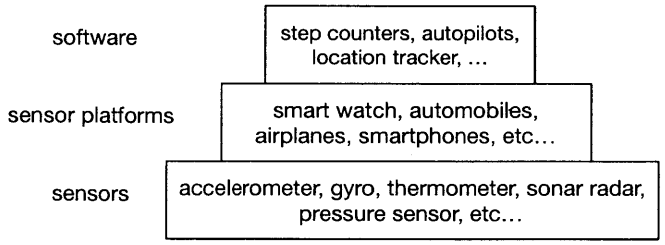


Figure 1-1: Sensor suppliers provide boards with sensors to hardware manufacturers who construct different sensor platforms that use those sensors. Those sensor platforms are often operated by software that use data read from sensors.

When sensors on embedded physical systems report anomalous data that violate these invariants due to unexpected noise or hardware failures, it can compromise the correctness of the systems built on top of those sensor readings. In fact, almost half of the accidents related to industrial chemical processes in France examined in one study have been due to errors in temperature and pressure sensor readings [18]. There are also examples in the aviation industry where aircrafts have crashed due to erroneous readings of pressure sensors [1].

Existing approaches to detecting anomalous sensor data in safety-critical applications involve selecting sensor data that is least likely to be corrupted, such as doing a majority vote from redundant sensors [22]. However, these approaches work for correcting failures of each sensor independently at the hardware level and ignores relationships between different sensors on a physical structure. As a result, if there is nothing wrong with the sensor hardware but the sensor readings violate the invariants that exist between sensor data or the constraints on sensor data values, simply correcting noisy sensor values is insufficient to guarantee robustness of the system. Often in an industrial setting dealing with hazardous chemicals, there is nothing wrong with sensor hardware themselves, but sensors report incorrect data because of poor installation, deficient maintenance, and poor cleaning [18]. Even when the sensors are reporting correct data that need special handling at the application level, if the software developer failed to handle the invariant violation, it could compromise safety of the system.

On the other side of the token, using unnecessary redundant sensors in non-safety critical applications can restrict the performance of the system by draining battery power and by taking up extra registers, memory, and CPU [8] [26] [27]. Especially on low power embedded systems with sensors, having to perform extra computations with redundant sensor data can create performance bottlenecks. If the sensor platform has, for example, two redundant accelerometers that are attached to the same physical body of the platform and constantly read a stream of correlated acceleration data, the hardware manufacturer can eliminate the usage of one of the accelerometers in order to trade off the reliability gained from using an additional set of accelerometers for the extra performance gained by freeing up resources used to read in another stream of data.

If there is a way to describe these invariants and make them available to a program written for computing on embedded physical structures with sensors, then the invariants could be exploited by the compiler of a sensor programming language in two ways. The first way is to generate code that provides implicit runtime assertions on the sensor data, effectively imposing constraints on what the sensor values could be at the software level, increasing the load on the CPU and other resources of the system for extra reliability (see Chapter 6.3). This approach is similar to how probabilistic asserts in [24] and how domain-specific languages like Matlab provide runtime assertions on numeric value ranges for certain mathematical operations. The benefit of having these assertions is reducing the burden on the programmer to catch silent errors of embedded system programs when some anomalous sensor data violate the invariants and the constraints on the hardware. Being able to catch these failures automatically enables the systems to fail fast and allows the programmer to focus their efforts elsewhere and develop more robust systems. It also gives more power to the programmer of the embedded systems to be able to handle those errors appropriately when the failures occur. The second way is to substitute code that does read/writes to redundant sensors with code that does reads/writes to only from one of the sensors by finding out, through the Newton description and another method of simulating program variable values (see Section 6.2.1), which of the variables hold values of redundant sensor readings. The benefit of this approach is reducing the overall code size and freeing up resources that would have been used to perform computation from all the redundant sensors.

To be able to describe the invariants that exist between sensor data on each embedded sensor system, the compilers of the embedded programming languages need to be able to take in an input configuration for each sensor platform that describe those invariants. Otherwise, the compilers have to be rewritten for each sensor platform to contain hard-coded information about those invariants. See Figure 1-2. This would mean that hardware manufacturers of the sensor platforms would need to delve into the compiler implementations of the programming languages dealing with the sensor platforms and then rewrite the portions of the compiler in order to hard-code each Physics data types needed for that sensor platform and the relationships between sensor data. In other words, if different sensor platforms use sensors that abide by different physical constraints, then multiple compilers of the same

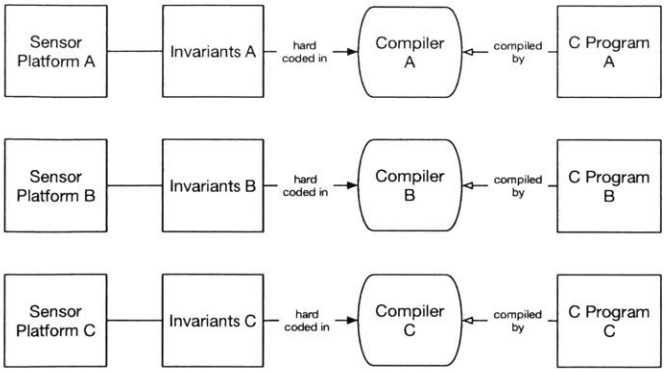


Figure 1-2: If C compilers wanted to generate code that takes advantage of the invariants that apply to each sensor platform, the compiler code that enforces the invariants would need to be changed for each sensor platform used.

programming language may be needed to accommodate those different situations. If there is a new language for the input configurations that describe the invariants for each sensor platform, embedded system programming language compilers can then use that information to be able to make transformations on their intermediate representations that take advantage of the invariants, without making significant changes to their own compiler implementations. See Figure 1-3.

Note that in Figure 1-1, the hardware manufacturers, represented by the middle layer of sensor platforms, are absent in situations where an independent programmer is developing software directly on top of the sensor layer. In this case, the programmer needs to write the Newton descriptions that contain the invariants appropriate for the use case of the sensors involved.

This MEng thesis introduces Newton, a language that describes constraints and invariants on values obtained from sensors embedded in physical structures and helps any compiler of another language generate code that enforces those invariants at runtime. The key contribution of Newton is that a description of a physical system written in Newton is able to describe invariants between sensor data. Once this file is parsed by the Newton compiler, that information is exposed through the Newton API, which consists of a compile-time library and a runtime library implemented as a part of this thesis. An application of the Newton libraries not implemented in this thesis involves another language compiler ("host language compiler") making calls to the Newton compile-time library. By using Newton,

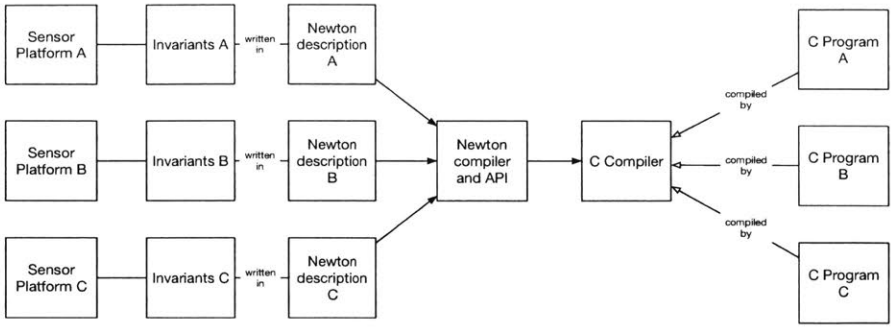


Figure 1-3: In contrast to Figure 1-2, the invariants that apply to each sensor platform are encoded in Newton descriptions which are input configurations to a C compiler. The C compiler code does not need to be changed to accommodate invariants in different sensor platforms because the Newton runtime library can be used at runtime with different Newton descriptions provide implicit assertions.

a host language compiler can verify that its variables have correct dimensional types and alter the code path in cases where the constraints described in the Newton description file are found to be violated at runtime by the Newton runtime library.

A Newton description file is written in a way that a simple physics formula sheet would be written (see Section 5.1). It is encapsulated by the Newton API and hidden from a host language compiler and, consequently, from the programmer of the host language. It should be noted here that the information gained from querying the Newton API does not benefit the programmer directly. The only agent calling the Newton API is a host language compiler.

To illustrate a violation of an invariant by anomalous sensor data, consider the following scenario. There is a weather balloon with pressure sensors, temperature sensors, a simple processor, and a small storage. The hardware manufacturer of this weather balloon intends this balloon to be only used in clear sky in an atmosphere that roughly follows Gay-Lussac's law of an ideal gas, $\texttt{pressure} = \texttt{constant} * \texttt{temperature}$, with some margin for tolerable noise in the sensors. Now, the invariant that the hardware manufacturer writes in the Newton description file would have the constraints of $\texttt{pressure} > \texttt{constant} * \texttt{pressure} - \epsilon$ and $\texttt{pressure} < \texttt{constant} * \texttt{pressure} + \epsilon$, where ϵ is some small number representing the margin for tolerable sensor noise. A programmer wants to write a software for the weather balloon that simply records readings from the pressure and temperature sensors as it floats in the sky. In the software, there exist variables `myPressure`, `myConstant`, and

myTemperature, where myPressure and myTemperature hold values of sensor readings from the pressure sensor and the temperature sensor, respectively, and myConstant is a constant. The values of these variables are to be written to the storage of the weather balloon. The data types of myPressure, myConstant, and myTemperature are floating point numbers. The hardware manufacturer of the weather balloon expects the values read from the sensors to match the theoretical description of Gay-Lussac's law throughout the runtime of the program. Initially, myTemperature is set to 2 after reading from a temperature sensor, and constant is set to 3. As a result, myPressure is assigned 6. Later, due to an unexpected noise in a pressure sensor, myPressure is set to 10, and no other variables changes their values. The compiler of a general-purpose programming language would not generate code that would raise an error at runtime because it does not understand that the variables in the code are supposed to be related to each other by Gay-Lussac's law. However, if the compiler of the above code understood that the variables are supposed to describe the weather balloon constrained by Gay-Lussac law, then the generated code should be able to catch the violation of this constraint at runtime when the value of myPressure was set to 10.

To be precise, we will say that a piece of code is completely mapped to Gay-Lussac's law (or any invariant) if at least the following four requirements are true:

1. The compiler of that program's language is able to pick Gay-Lussac's law as the invariant that describes the relationship between the variables myPressure, myConstant, and myTemperature out of all the invariants available to the compiler. Newton allows the hardware manufacturer to write an input configuration file of invariants for a particular sensor platform and to enable other compilers to be able to pick the invariant relevant to a set of variables in the program. See Figure 1-4. The invariants in Newton will involve the typical sensors available on commercial hardware platforms that measure the following quantities: acceleration, humidity, temperature, pressure, gyroscopes, luminosity, and magnetic flux. See Section 6.2.3 for details of how Newton accomplishes this requirement.

2. There is a mapping between the variables in the host language program and the pa-

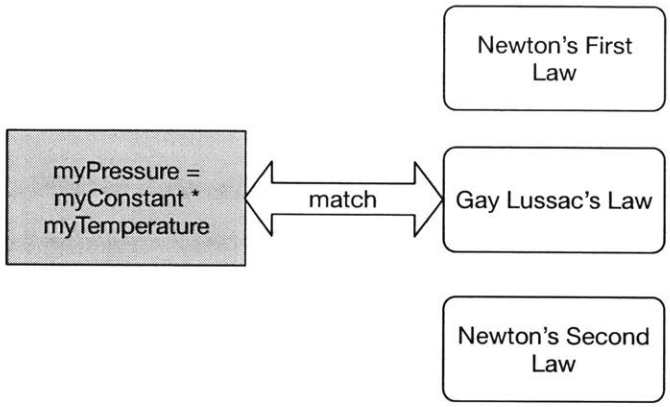


Figure 1-4: The first step is to pick the invariant that describes a given program from all invariants available to the compiler of the embedded programming language. These invariants can be written in a Newton description.

rameters of an invariant. Specifically, `myPressure` in the program needs to describe `pressure`; `myConstant`, `constant`; and `myTemperature`, `temperature`. See Figure 1-5. This step is achieved in two ways. The first way is to provide additional type information in Newton descriptions (see Section 5.1.1) and to infer the mappings between variables and parameters by simulating sensor values and variable values (see Section 6.2.1). The alternative approach is for the host language to provide an annotation syntax that marks a particular function as a routine that returns values to be checked for an invariant. The two approaches and their advantages are discussed in Sections 6.2.1 and 6.2.2.

3. The compiler of the embedded programming language uses separate data types for `myPressure`, `myConstant`, and `myTemperature` to accurately reflect their physical types. See Figure 1-6. A description written in Newton provides information about the physical types that can be used by the compiler of the embedded programming language. After satisfying requirement 2, `myPressure`, `myConstant`, and `myTemperature` are all floating point numbers, but `pressure`, `k`, and `temperature` are different physical types. One way to meet this requirement is to identify the dimension type of each variable. For example, `myPressure` would be of data type `pressure` with units of `Pascal`. This step has been the topic of research in the past [21] [14], as will be discussed in Chapter 2. Although satisfying requirements 1 and 2

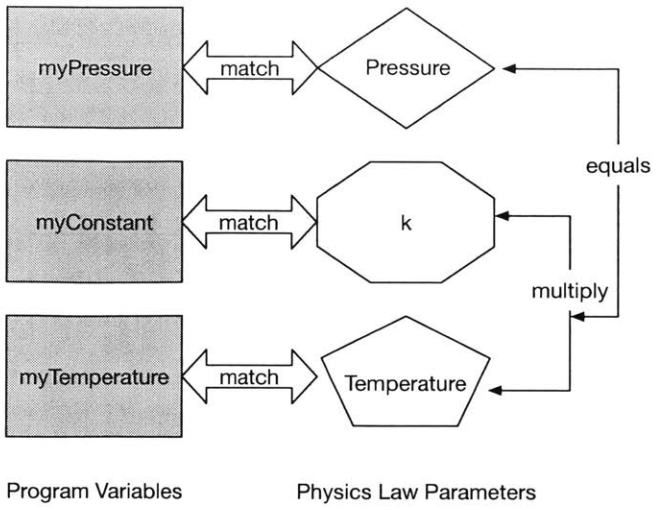


Figure 1-5: The variables of a program need to be matched with the parameters of an invariant. The black arrows indicate constraints between the parameters of the invariant, and the white arrows indicate the mapping between the variables of the program and the parameters of the invariant. The uniform shapes on the left hand side show that the program variables are represented by a single data type (float) while the different shapes on the right hand side show that the parameters are of different physical types.

is sufficient to calculate the numerical value of `myPressure`, but without satisfying requirement 3, the program variables would lack the descriptions of the physical types of Gay-Lussac's law parameters. See Section 5.1.1 for details of how Newton accomplishes this requirement.

4. The numerical values of `myPressure`, `myConstant`, and `myTemperature` should satisfy the invariant with constraints `pressure` $>$ `constant` $*$ `pressure` $- \epsilon$ and `pressure` $<$ `constant` $*$ `pressure` $+ \epsilon$ throughout the runtime of the program. If, at a later runtime of the program, the numerical value of `myPressure` changes but the numerical values of `myConstant` and `myTemperature` do not, then the constraint described by Gay-Lussac's law is then violated, and this program can no longer be said to satisfy this invariant. See Figure 1-7. Information provided by the Newton API can then be used by a host language compiler to transform the original source code AST in a way that improves reliability or performance of the source code. See Sections 6.2 and 6.3 for details about the transformations.

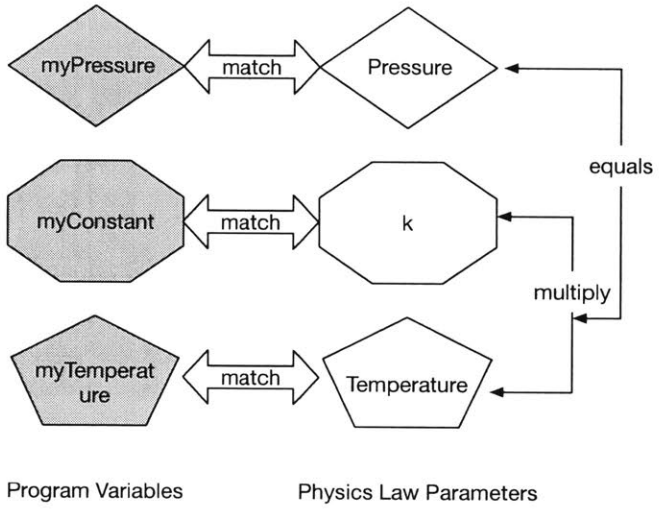


Figure 1-6: The data types of the variables need to reflect the physical types of the parameters of the invariant. In Figure 1-5, the data types of the variables in the program are identical when in reality they represent inherently distinct physical concepts. This step provides different data types for variables that represent different physical quantities.

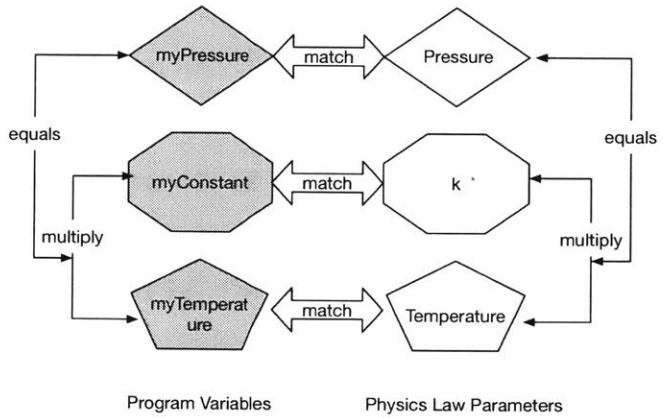


Figure 1-7: The values of the variables need to constrained to each other in the same way the parameters of the invariant are.

The research contributions of Newton are on requirements 1, 2, and 4 stated above, and they are described in Chapters 5 and 6 of this thesis. The Newton API allows another compiler to bind variables of a host language to a constraint and its parameters. As a part of this MEng thesis, the Newton compiler and the Newton API are implemented, but a host language compiler that would use the Newton API is not. We present algorithms of how a host language compiler should interact with the Newton API to make necessary transformations to its source code and example scenarios in the appendix that benefit from using the Newton API.

Chapter 2

Background

In the theoretical domain, much work has been done in dimensional analysis by Buckingham[17], Brand[16], and Kurth[23], who made contributions to the Buckingham Pi Theorem, which states that any constraint between n physical quantities subsisting k independent basic physical quantities can be represented by a system of $n - k$ linearly independent equations. One of the advantages of using the Buckingham Pi Theorem was that fewer parameters could be considered for numerical analysis or experimentation to investigate a relationship between them. In fact, Stoutemeyer developed a program implemented in MACSYMA that check dimensions of other programs by using the results from the Buckingham Pi Theorem [28].

In programming languages, introducing dimension types has been explored through built-in types and library calls. For example, House has proposed extending the Pascal language with units and dimensions[19], and F# includes ways for the programmer to declare any dimensions of physical quantities and use them in code [21]. F# provides support for type checking and type inference based on the dimension unification algorithm described in [20]. XeLda provides type checking for data in Excel spreadsheet programming and automatically generates constraints if there are circular references [14].

Newton is the only system to our knowledge that allows a host language compiler to bind programming variables to invariants. A host language compiler can accomplish this binding by inserting new code into the intermediate representation that would query the Newton API with its variables. With the information gained from the Newton API, a host

language compiler is able to modify its intermediate representation in a way that makes the original code more robust to errors or improve its performance (see Sections 6.2 and 6.3).

Newton is different from F#, XeLda, Pascal, and other languages' dimension type systems in that it provides another programming language a capability to do more than dimension type checking. Newton can check whether invariants between sensor values are preserved. On the other hand, Newton is more restricted than languages like F# and Pascal because Newton is only meant to describe sensor platforms. Refer to Section 6.2.1 to see how Newton binds variables that are only directly correlated with sensor types and therefore is most suitable for describing sensor platforms.

Chapter 3

Research Questions and Aims

The goals of this MEng thesis are:

1. To implement a Newton compiler that correctly parses physics constraints and an API that returns correct units of various Physics types. See Section 6.1 to see how Newton meets this goal.

2. To implement a Newton API that can be queried by a host language compiler and can perform type checking on an expression made of Physics types. See Section 6.1 to see how Newton meets this goal.

3. To investigate if it is possible to describe real-world physical systems with Newton. See Appendix A for example systems that might benefit from Newton.

4. To describe an algorithm that a host language compiler can follow to perform dimensional type checking using the Newton API. See Section 6.1

5. To describe algorithms that a host language compiler can follow to make transformations to its IR using the Newton API, in order to make the original source code more reliable or improve its performance. See Sections 6.2 and 6.3.

6. To describe how programs on real-world physical systems may use the transformations. See Appendix A.

The above goals are completed at the writing of this thesis. To see items that have yet to be completed for this project, see Chapter 8.

Chapter 4

Newton Language by Examples

Information within a Newton description is accessible by a collection of methods, the Newton API, consisting of a compile-time library used by a host language compiler and a runtime library which is linked against the host language program.

A host language compiler makes a call to methods in the Newton compile-time library to make sure that its variables abide by dimension constraints. In addition, the host language compiler can inject code into its IR so that the generated code would call the Newton API at runtime to check certain variables abide by value invariants.

The purpose of the Newton compile-time library is to check if an expression or an assignment statement in a host language program is dimensionally consistent and to transform the host language program in a way that takes advantage of the invariants written in the Newton description. the purpose of the Newton runtime library is to check if a set of variables in a host language program abide by value invariants. The transformed host language program would then make a call to the Newton runtime library to check its variable values against Newton invariants at runtime. See Figure 4-1 for an overview of how Newton interacts with a host language compiler.

In this section of the thesis, we take a look at an example as an overview of how the Newton compile-time library may be used by a host language compiler to perform dimensional type checking and an IR transformation that result in a code that checks invariants on host language variables at runtime, using the Newton runtime library. Later parts of the thesis describe more details and another IR transformation that reduces sensor redundancy in a host

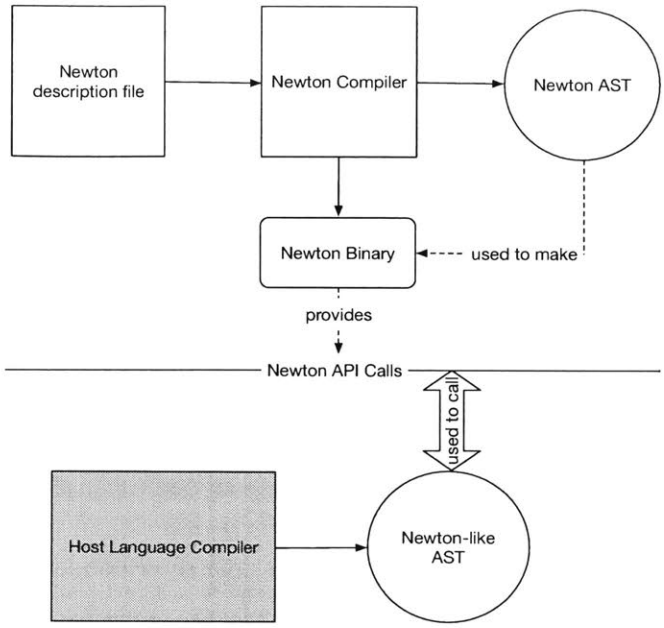


Figure 4-1: A Newton description file is used by the Newton compiler to generate an AST, and a host language compiler also generates a Newton-like AST of its own code. The Newton API is then used to make sure that the host language compiler's AST matches the constraints embedded in the Newton compiler's AST.

language program. Suppose we have a small device with an accelerometer sensor attached to a simple pendulum and a program that reads values from the device to perform some computation. In addition, suppose that the hardware manufacturer specified in a Newton description the following constraints on acceleration values.

```
SimplePendulum : invariant(a: acceleration) = {
    a >= 3 * m / s ** 2,
    a <= 9 * m / s ** 2
}
```

This code describes an `invariant` which is a collection of constraints that apply to the parameters given by the caller of the Newton runtime library, in this case, a variable named `a` of Physics type `acceleration`. The constraint states that the numeric value bound to the variable of type `acceleration` must be between 3 and 9 and that the dimension value bound to the Physics type `acceleration` of the variable `a` must equal m/s^2.

Now, consider the following code written in C that assumes a compiler which interacts with Newton. The code was written to count the swings of a pendulum with accelerometers by detecting sign changes in the accelerometer data and has an additional syntax to C in

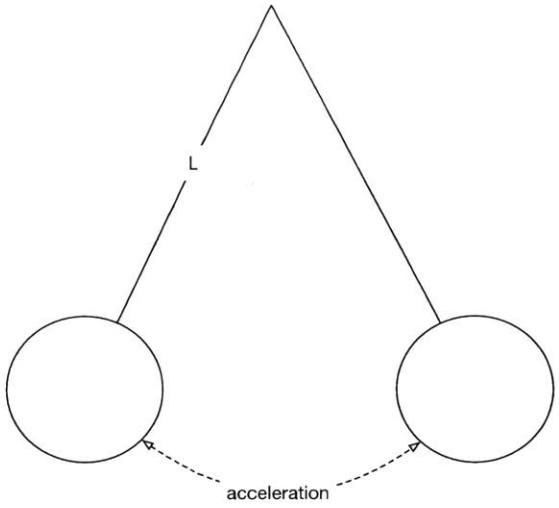


Figure 4-2: A simple pendulum

order to define Physics types.

```
acceleration@0 prevXacceleration = readFromXAccelerometer();
time durationInSeconds = 1000 * 30; // thirty seconds
time startTime = readFromSystemClock();

int swingCount = 0;

while (readFromSystemClock() < startTime + durationInSeconds)
{
  acceleration@0 xAcceleration = readFromXAccelerometer();

  if (prevXacceleration * xAcceleration < 0) {
    swingCount++;
  }

  if (xAcceleration != 0) { // do not double count the change
    prevXacceleration = xAcceleration;
  }
}

printf("detected %d swings in the pendulum\n", swingCount);
```

Notice that in the places where primitive type identifiers belong, we instead have the strings "acceleration" and "time". These strings "acceleration" and "time" represent a new kind of data types not present in the original set of C data types. They represent the physical concepts of acceleration and time as new data types ("Physics types") as defined in a Newton description that the C compiler takes in as an input. The "@" indicates that the acceleration type is in the first sub-dimension, which is the x axis in this context. Having these new

data types is enabled by a language extension or a pragma directive that would indicate to the C compiler that "acceleration@0" means an additional data type. Having these data types satisfies the requirement 3 mentioned in Section 1. Before the C runtime can call the Newton runtime library method `newtonApiSatisfiesConstraints` to check invariants are satisfied for its variables, the C compiler needs the following actions performed, using methods in the Newton compile-time library.

1. Compile the Newton description given by the sensor platform manufacturer.

2. Perform compile-time dimensional type checking. This step requires building a Newton AST of the C program that would be passed into by the Newton libraries to perform dimensional type checking at compile-time or to check invariants at runtime. If the Newton library methods operated on host language compiler IR's, building a Newton AST would not be necessary, but having a standard Newton AST is what enables the Newton libraries to interact with different host language compilers.

3. Select variables declared or used in a block, where a block is a function scope or a loop scope, that are parameters of invariants in the Newton description. This step involves requirements 1 and 2 mentioned in Chapter 1.

4. Transform the C source code (i.e. C compiler's IR) to call `newtonApiSatisfiesConstraints` for appropriately selected variables for every block.

The first step initializes the Newton API with the relevant Newton description. The second step of dimensional type checking ensures type safety at compile-time before values of host language programs are checked at runtime. The third step selects parameters out of the host language program variables to pass into the API method `newtonApiSatisfiesConstraints`. The fourth step enables the host language compiler to produce code that calls `newtonApiSatisfiesConstraints` at runtime. Now, we examine the above steps more closely.

The first API call made, `newtonApiInit`, is a call to the Newton lexer and parser to read the given Newton description file. This call stores all information about Physics types

and invariants of the given Newton description in memory so that the C compiler can call methods of the Newton API.

```
State * newton = newtonApiInit("/file/path/to/invariants.nt");
```

The next action that needs to be performed is compile-time dimensional type checking, and the C compiler needs to construct the Newton's version of its AST before it can perform dimensional type checking because dimensional type checking at compile-time and invariant checking at runtime in Newton operate on their input Newton AST's. Requiring Newton library methods' inputs as Newton AST's enables Newton to work with different host language compilers.

The Physics types remain in the background as a numeric type in the C compiler's intermediate representation, but at compile-time the Newton version of the C compiler's intermediate representation is used to perform dimensional type checking (see Section 5.2 for more detail). A numeric type could be int, float, double, or whatever data type that a host language has available in its built-in types to represent the numeric values of sensor data. As such, all data types not recognized by the original C syntax but defined in the Newton description will be treated as a Physics type only for the purpose of calling the Newton API methods, but the original C intermediate representation will keep those variables as numeric types. As mentioned in Chapter 1, having these data types is enabled by language extensions or pragma directives that would indicate to the host language compiler that the tokens not recognized in the original syntax of the host language represent Physics types.

Now we need to construct a Newton AST of the C program. The Newton AST of a host language program would comprise Newton nodes that represent host language constructs, just in the format of the Newton AST. For example, an identifier in C is represented by `kNewtonIrNodeType_Tidentifier` in Newton, and an add operation + is represented by `kNewtonIrNodeType_Tplus`. The code below shows how to initialize Newton identifier nodes for the pendulum program variables `prevXacceleration` and `xAcceleration`. These identifier nodes are one of the node types that constitute the leaf nodes of the Newton AST, and other nodes are initialized in the same way. To see how expression trees and statement trees are constructed out of identifier nodes, other factors, and term nodes, refer to Section 5.2.1.

```
/*
 * Makes a Newton identifier node of type "acceleration"
 * with value 5.
 */
NoisyIrNode * accelerationNode = makeNoisyIrNodeSetToken(
    noisy,
    kNewtonIrNodeType_Tidentifier,
    "acceleration",
    5.0
);
```

For every Physics type token in the C program, such as `acceleration`, the C compiler needs to look up the type using the Newton compile-time library. The call to `newtonApiGetPhysicsTypeByName` with the given string "acceleration" and the Newton state returns a `Physics` struct with identifier `acceleration` that the caller uses to set the `physics` field of the struct `IrNode`, which is used to build a Newton's parallel of the C compiler's AST. The `physics` fields of `IrNodes` are compared during dimensional type checking. The same set of calls happen for the string "time." See Section 5.2.1 for more details.

```
accelerationNode->physics = newtonApiGetPhysicsTypeByName(newton, "
    acceleration");
```

Some Physics types may exist in more than one dimension. For example, `acceleration` can exist along x, y, and z dimensions, and there might be more than one `temperature` sensor in a sensor platform. This is denoted by the symbol "@" as in "acceleration@0" which indicates that this Physics type exists in the first dimension, in this case the x axis. If a sensor platform had 2 different temperature sensors, the Newton description would specify 2 different temperature types of "temperature@0" and "temperature@1". The C compiler can then make a call to `newtonApiGetPhysicsTypeByNameAndSubindex` with a particular subindex to set the type to the Physics struct of `accelerationNode` to an `acceleration` in the x axis.

```
accelerationNode->physics = newtonApiGetPhysicsTypeByNameAndSubindex(newton,
    distanceNode->token->identifier, 0);
newtonApiAddLeafWithChainingSeqNoLexer(newton, root, accelerationNode);
```

After these calls on type expressions of variables, the C compiler needs to make some more API calls to perform type checking on assignment statements and expressions. If the following properties are maintained throughout the statement or expression tree, then the

Newton API call `newtonApiDimensionCheckTree` will tell the caller that the entire tree is dimensionally consistent by returning a Boolean.

1. Binary operations of multiply and divide can operate have operands of different Physics types.

2. Binary operations of add and subtract must have operands of same Physics types.

3. The exponent of an exponential expression must be dimensionless.

4. Compare operations such as $>, <, <=, >=$ must have operands of same Physics types.

5. Assign operations such as $=$ must have operands of same Physics types.

The code below shows a Newton compile-time library call that takes in a statement or an expression Newton AST and checks if the properties above are maintained in this tree.

```
newtonApiDimensionCheckTree(newton, root);
```

For more information about the rules of dimensional consistency, see Section 5.3.1. We also describe the algorithm to check dimensional consistency of the entire C program more in detail in Section 5.3.1.

At this point, suppose that the Newton AST of the C program is constructed, and all statements and expressions in the program are dimensionally consistent.

The next step is to select the variables that are parameters of the invariant SimplePendulum reproduced here.

```
SimplePendulum : invariant(a: acceleration) =
{
    a >= 3 * m / s ** 2,
    a <= 9 * m / s ** 2
}
```

The C compiler needs to select variables from its previously constructed Newton AST of its program (see Section 6.2.3). As mentioned in Chapter 1, this step can be achieved in two ways. The first way is to match the types of the parameters of the invariants to the the types of the variables and to infer the mappings between simulated values of the variables and the sensor readings (see Sections 6.2.1 and 6.2.2). The second way is for the host language to

provide a syntax where the programmer can annotate functions that return values that should be checked by an invariant. Host language program variables whose values are assigned from calling these annotated functions are marked as a potential parameter to be passed into a Newton invariant (see Section 6.2.2).

In the C program we are considering, there are only two variables of type `acceleration`, `prevXacceleration` and `xAcceleration`, and the invariant SimplePendulum takes in a single parameter of type `acceleration`. Thus, those two variables are independently selected as parameters of SimplePendulum.

The caller then needs to number their parameters from 0 to $n-1$, where n is the number of parameters passed into the Newton API call. Numbering the parameters sets the order of the parameters as defined in the invariant scope of the Newton description and effectively allows the Newton API to distinguish between two variables of identical Physics type (see Section 6.2.4).

```
IrNode *  root = genIrNode(newton,  kNewtonIrNodeType_PparameterTuple,
  NULL /* left child */,
  NULL /* right child */,
  NULL /* source info */);
  IrNode* parameter = deepCopyNode(accelerationNode);
  parameter->type = kNewtonIrNodeType_Pparameter;
newtonApiAddLeaf(newton, root, accelerationNode);

newtonApiNumberParametersZeroToN(newton, root);
```

Finally, the Newton runtime library call `newtonApiSatisfiesConstraints` can be inserted into the C compiler's intermediate representation with the parameter trees we constructed so that the generated code will call `newtonApiSatisfiesConstraints` that performs value checking at runtime. The Newton runtime library must be linked against the C program before it is run. Changing the C compiler's intermediate representation is not part of the Newton API, but an application of the Newton API (see Figure 5-1).

At runtime, the result of the call `newtonApiSatisfiesConstraints`, the `NewtonAPIReport` struct, will show that the the constraints passed value constraints since 5 is between 3 and 9.

```
NewtonAPIReport* newtonReport = newtonApiSatisfiesConstraints(
    newton,
    parameterRoot
);
```

The runtime call `newtonApiSatisfiesConstraints` also checks for dimensional consistency to evaluate the dimension of exponential expressions (see Section 5.3.2), and it satisfies since `acceleration` defined in the Newton description `invariants.nt` has dimensions m/s^2.

Now that the code to check if invariants are satisfied is included in the C program (C intermediate representation), the C compiler can perform transformations to its program. One transformation that a host language compiler can perform, covered in Section 6.2, executes a global error handler if the invariant was found to be violated at runtime.

The transformation is the following. For every set of variables that matched any Newton invariant, the C runtime calls the Newton runtime library routine `newtonApiSatisfiesConstraints` before first usage of any of the matched variables but after all of the matched variables have been defined and assigned a value, and then executes a block of code annotated by the programmer to be the global error handler function. The transformation repeats the steps above for every variable set that matches any Newton invariants in the Newton description. Calling the Newton API before the first usage of matched variables after they have been defined checks if those variable values satisfy the Newton invariant.

A precondition of this transformation is that the C intermediate representation should be in the Single Static Assignment form, in which every variable of the C intermediate representation is assigned a value exactly once and is defined before used. Using the Single Static Assignment form simplifies the problem of finding every usage of the matched variables after they are assigned a value, otherwise done via use-def chain analysis. This requirement is useful for being able to insert the call to a global error handler before the matched variables are used. In addition, requiring the SSA form forces the host language compiler to declare variables that directly represent sensors but are only used to define other variables. For example, in the code "`foo = read_from_acceleration_x * 5`", `read_from_acceleration_x` would be declared in another variable `bar` before being used to define `foo`. This way, even though a host language program may not directly use the variables that contain sensor data, the host language compiler can still check invariants on sensor data at runtime if any variable is assigned a value computed from sensor data.

Shown below is the transformed version of the pendulum swing counting code, where every variable is assigned a value exactly once and defined before used. Note that the sensor platform used in this code has an accelerometer, and the code shown below is the body of the main function. To facilitate presentation, only the variables that would be passed into the Newton runtime library routine are shown to be declared.

```
1  // custom-made Boolean that contains invariant checking result for this block
2  bool VALID = true;
3
4  // matches SimplePendulum invariant
5  acceleration@0 prevXacceleration1 = readFromXAccelerometer();
6
7  time durationInSeconds = 1000 * 30; // thirty seconds
8  time startTime = readFromSystemClock();
9  int swingCount = 0;
10
11 while (readFromSystemClock() < startTime + durationInSeconds) {
12     // custom-made Boolean that contains invariant checking result for this
       block
13     bool VALID = true;
14
15     // matches SimplePendulum invariant
16     acceleration@0 xAcceleration = readFromXAccelerometer();
17
18     prevXacceleration2 = PHI(prevXacceleration1, prevXacceleration3);
19
20     // Call Newton API before the first usage of prevXacceleration2.
21     NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
       parameter tree made of prevXacceleration1 */);
22     VALID = VALID && report->satisfiesValueConstraint;
23
24     // if invariant violated.
25     if (!VALID)
26         globalErrorHandler(report);
27
28     // Call Newton API before the first usage of xAcceleration.
29     VALID = VALID && newtonApiSatisfiesConstraints(newton, /* parameter tree
       made of xAcceleration */)->satisfiesValueConstraint;
30
31     // invariant violated
32     if (!VALID)
33         globalErrorHandler(report);
34
35     // first usage of prevXacceleration2 and xAcceleration
36     if (prevXacceleration2 * xAcceleration < 0) {
37         swingCount++;
38     }
39
40     if (xAcceleration != 0) { // do not double count the change
41         prevXacceleration3 = xAcceleration;
42     }
43 }
44
```

```
45 printf("detected %d swings in the pendulum\n", swingCount);
```

Now, we explain how the pendulum code was transformed line by line. Line 2 and 13 introduce a custom Boolean variable for the function block and the loop block. These Booleans will store the result of the runtime library call `newtonApiSatisfiesConstraints`. Line 5 has a variable named `prevXacceleration1` which holds the numeric value of the x-axis accelerometer and thus will be considered a parameter to be passed into `newtonApiSatisfiesConstraints`. Variables in line 7 and 8, `durationInSeconds` and `startTime` do not hold direct sensor values throughout the runtime of the program, as determined by the method called dynamic tagging we describe in Section 6.2.1, and will not be a parameter to the runtime call. Line 16 has a variable named `xAcceleration` which would also hold the numeric value of the x-axis accelerometer and thus will be a parameter to the Newton runtime call. Line 18 uses an operation `PHI` to initialize the variable `prevXacceleration2` from `prevXacceleration1` or `prevXacceleration3`, depending on whether the control flow came from outside the loop or from the previous iteration of the for loop.

Line 36 is the first time inside the for loop that the variables `prevXacceleration2` and `xAcceleration` are used, so before Line 36, this transformation inserts code that does 3 things. First, not explicitly shown in this code, it selects out of the variables that are declared or used in the current loop - `xAcceleration`, `prevXacceleration2`, `durationInSeconds`, `startTime`, and `swingCount` - parameters that should be passed into the Newton runtime call `newtonApiSatisfiesConstraints` by running dynamic tagging and matching types, which are `prevXacceleration2` and `xAcceleration`. Second, also not explicitly shown in this code, it constructs Newton parameter trees as illustrated previously. Third, Lines 21 and 29 introduce a call to `newtonApiSatisfiesConstraints` with the parameter trees created. Lines 25 and 32 execute the global error handler if the invariant was violated any point so far. Note that the global error handler is not related to Newton in any way — the host language must provide a syntax for the programmer to designate a function as the handler, either through annotations or reserving a particular function identifier as a keyword that denotes the global error handler. The error handler function can take in a `NewtonAPIReport` struct to provide better error messages. This

approach is similar to how exceptions are thrown in Java.

The following code shows a hypothetical annotation syntax and a global error handler.

```
@@error
void globalErrorHandler(NewtonAPIReport* report) {
    ConstraintReport* currentConstraint = report->firstConstraintReport;
    while (currentConstraint != NULL) {
        /* log errors in some file */
        currentConstraint = firstConstraint->next;
    }
}
```

Calling the Newton API before the usage of sensor variables after every assignment statement ensures that invariants are checked on those invariants before those variables are used in further computation. Because of the fact that there may be multiple variables of the same Physics type that can match invariants, there are limitations to this method, which we discuss in Section 6.2.7.

The overhead of using Newton is extra runtime required at compile-time and at runtime to call the library methods. The below table summarizes the averages runtimes (in nanoseconds) of the two most complex Newton API calls `newtonApiInit` and `newtonApiSatisfiesConstraints`. For each Newton description, the methods are run 40 times.

Newton Description	newtonApiInit	newtonApiSatisfiesConstraints
Step Counter	1503298	139392
Activity Classifier	1339013	156423
Vehicle Distance	675434	38847
Weather Balloon	1032082	89110
Airplane Pressure	684599	51114
Ball Dropped	1027406	77935
GPS Walking	786148	60469
Jet Engine	1010725	109836
Motor Wheel Chair	972281	63757
Reactor Rod	68395	62814
Sensor Life	1006878	88640
Tire Pressure	1621634	142686

The rest of this thesis explores the implementation of the methods in the Newton compile-time library and the Newton runtime library as well as their applications.

Chapter 5

Newton Design

In this chapter of the thesis, we examine different components of the Newton API implementation: Physics data types, the syntax of the Newton description, constructing the Newton parallel of the host language compiler's AST, and walking those trees for type checking and value comparison. The next chapter builds on the content of this chapter and covers how to use the Newton AST in order to perform dimension type checking and value checking, and finally, how to transform the source code by inserting the call to the Newton runtime library routine `newtonApiSatisfiesConstraints` into the host language compiler's AST.

5.1 The Newton Description File

One of the core ideas behind the design of Newton is that a designer of hardware can plug a Newton description file into the host language compilation process in order to specify custom laws of physics as we saw in Section 4. See Figure 1-3 and Figure 1-2.

As explained in Chapter 4, Physics data types are additional data types to the original set of data types in a host language, enabled by language extensions or pragma directives. Having these Physics data types allows the programmer to declare variables or functions in terms of those data types.

For example, the programmer can declare a variable as an `acceleration` type with a numerical value 5.

```
acceleration foo = 5;
```

A Newton description contains all the Physics data types available to a host language program and invariants between sensor data during the host language compilation process. Thus, a host language compiler can use a different Newton description to make a different set of Physics data types and invariants available for a host language program.

The following two sections describe Physics data types and the structure of the Newton description. The formal grammar of the Newton description is described in Appendix Section B.

5.1.1 Physics Types in Newton

As mentioned in Chapter 2, being able to create types that represent quantities in Physics is not a novel feature. Newton implements basic type checking of expressions and statements of a host language program at compile-time similar to F# [21]. In F#, a programmer can declare a custom type and declare variables in that type.

```
1 [<Measure>] type m   (* meter *)
2 [<Measure>] type s   (* time *)
3 let distance = 100.0<m>
4 let time = 5.0<s>
5 let speed = distance / time
```

Note that the type for speed is statically derived to be m/s by F# compiler. In F#, static type checking uses these custom types defined by the programmer, but during runtime the types of variables in F# are set to a numeric type.

Type checking in Newton is similar to F# in that custom types can be declared that represent Physics quantities, but it is different from F# in that the hardware manufacturer, not the programmer (of a language whose compiler interacts with Newton), declares custom data types available to a host language program by writing a Newton description as an input to the host language compiler (see Figure 4-1). In F#, the programmer writes additional code to define custom types in that program, but in Newton, custom types are already defined in a Newton description, which is used in a host language compilation process. Having this capability to define custom types allows type-checking of statements and expressions of a host language program through the Newton API. The following describes how these custom Physics types are declared in a Newton description.

1. Newton allows assigning dimensions to "base" Physics types. The base Physics types in Newton are the simplest building blocks that make up other derived Physics types. For example, the unit of seconds "s" describes the dimension of `time`.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

distance : signal(i: 0 to 2) = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}
```

2. In Newton, "derived" Physics types can be defined in terms of base Physics types or other derived Physics types, thereby assigning dimensions to the derived Physics types. For example, `speed` is `distance` divided by `time`. The Newton compiler computes the dimension of `speed` based on the base Physics types defined or on the previously defined derived Physics types.

```
speed : signal(i: 0 to 2) = {
    derivation = distance@i / time;
}
```

   This feature also allows definition of physics constants.

```
speedLimit: constant = 100 * m / s;
```

3. For some Physics types defined, Newton can define the axes at which they occur. For example, `distance`, `speed`, and `acceleration` have 0, 1, and 2 axes which can be interpreted as x, y, and z axes.

5.1.2 Syntax of the Newton Description

This section explains where these Physics types are defined in the Newton description and how they are used to define invariants. There are the three components of a Newton description file: signal scopes, constant statements, and invariant scopes.

1. Signal scopes define either base or derived Physics types and their units. Base Physics types have `none` as their derivation, and derived Physics types have expressions of

base Physics types or other previously defined derived Physics types. These Physics types represent the signals read from sensors on a hardware platform. Some base Physics types exist primarily to be able to define derived Physics types of typical sensors on a commercial hardware, such as acceleration, magnetic flux, and gyro, where other base Physics types directly represent the signals of sensors, such as the temperature sensor.

2. Invariant scopes define a set of constraints that must be preserved for a set of parameters. A parameter is a host language program variable with the specified Physics type in the Newton invariant signature. All constraints defined in an invariant scope must hold true for all of the parameters in that scope. Note that the Newton libraries do not select which host language program variables should be described by Newton invariants, but the host language compiler does. Invariant scopes are similar to functions in a way that they take in a list of parameters with designated types and specifies some logic that applies to the parameters. See the next subsection and Section 5.3.2 for more details.

3. The constant statements define physics constants, such as Newton's gravitation constant, with their corresponding dimensions expressed in terms of the base dimensions. The constants can also be dimensionless, such as the mathematical constant π.

4. In derived Physics types, `name` and `symbol` can be aliases for the SI units. For example, `Force` can have either $kg * m/s^2$ or N, where $kg * m/s^2$ comes from Newton compiler derivation and N is the symbol of the name `Newton`. This approach is similar to aliasing in F# [21].

The following is an example Newton description that illustrates the three components described above.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

distance : signal(i: 0 to 2) = {
    name = "meter" English
```

```
    symbol = "m";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

speed : signal(i: 0 to 2) = {
    derivation = distance@i / time;
}

acceleration : signal(i: 0 to 2) = {
    derivation = speed@i / time;
}

Pi : constant = 3.14159;
g : constant = 9.8 * meter ** 2 * second ** -2;
pendulumLength : constant = 3 * m;

PendulumInvariant : invariant(
    a: acceleration@0,
    period: time
) = {
    a >= 2.2 * m / s ** 2,
    a <=  10 * m / s ** 2,
    period ~ (4 * Pi * Pi * pendulumLength / g) ** 0.5,
    period >= 3 * second,
    period <= 9 * second
}
```

This Newton description defines base Physics types of `distance` and `time` and derived Physics types of `speed` and `acceleration`. `PendulumInvariant` is an invariant that takes in an x axis acceleration and a variable whose value is the period of the pendulum, and this invariant asserts some simple properties that must be preserved for this particular sensor platform.

5.2 The Newton AST

The main two applications of Newton API are contributing to compile-time checks performed by the host language compiler and modifying the host language source code using information from the Newton API in a way that makes the original source code written in the host language more robust or improve its performance. The Newton AST is the

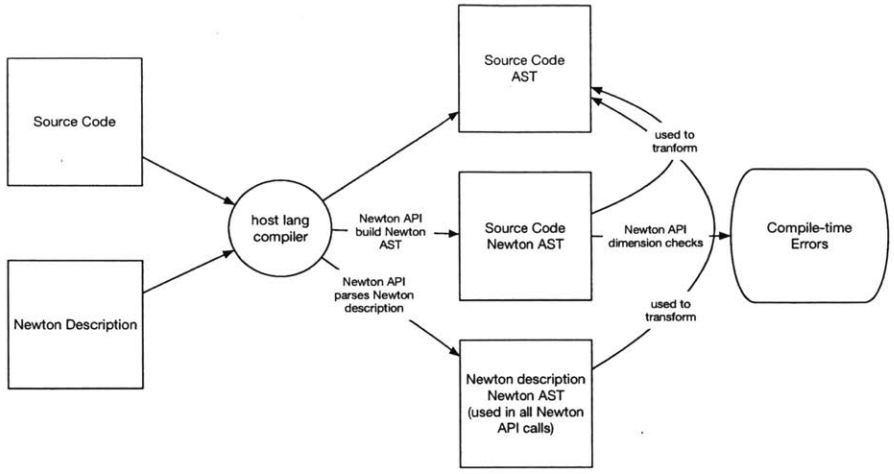


Figure 5-1: Two different Newton AST's are created from source code and the Newton description. The Newton AST of the Newton description is used built in the Newton API call `newtonApiInit` and helps build the Newton AST of the source code as well as type-check it. The source code Newton AST is then used to construct parameters that would be passed into the invariant checking API, which reads invariant subtrees of the Newton AST of the Newton description. The code to perform this checking and optimization is incorporated into the host language compiler's AST.

main interface through which type checking of the source code and its transformation are achieved. See Figure 5-1. There are two different Newton AST's involved, one representing the Newton description and the other representing information about portions the host language program which are passed to the Newton compile-time library for validation.

From the API caller's perspective, the host language compiler builds Newton AST's, in addition to its own intermediate representation for portions of its programs that need to be validated by the Newton libraries. Specifically, the Newton AST's are used to call `newtonApiDimensionCheckTree` which walks the input Newton tree of a statement or an expression to perform dimension type checking, and they are also scanned to build lists of parameters to pass into `newtonApiSatisfiesConstraints` as discussed in Section 6.2.3.

The Newton API then validates the input from the host language compiler, which is given in the form of a Newton tree, with information contained in the Newton description, which is encoded in the form of Newton AST's and other symbol tables. In `newtonApiDimensionCheckTree` (dimension type checking), the Newton API simply walks the host language compiler's Newton AST (built by calling meth-

ods in the Newton compile-time library as we will see in the next section), and in `newtonApiSatisfiesConstraints` (runtime value checking and type checking), the Newton API walks subtrees in its own Newton AST (more specifically, a single invariant scope subtree) of the Newton description and looks up parameters passed in from the host language runtime. Requiring the input trees to be in the form of Newton AST unifies the interface between the Newton API and many different host language compilers.

As of the current implementation, the Newton language only has simplest language constructs to enable a host language compiler to build its Newton AST of only simple blocks (in this thesis, function scopes and loop scopes) containing statements and expressions, that is programs without branches. However, even for a complex program, a host language compiler can still build small separate trees of its statements and expressions for dimension type checking since the Newton API takes in a statement or an expression Newton tree. This way, the host language can have language constructs not defined in Newton and still utilize the Newton API to perform checks on its source code.

5.2.1 Building Newton AST's for Newton Descriptions and for Host Language Expression and Statements

To see the node types currently supported by the Newton API, refer to Appendix Section D. Building the Newton AST for the Newton parser and for the host language compiler is identical except for how to set the Physics types of the identifier nodes.

5.2.2 Building the Newton AST: Creating Newton AST Nodes

Creating a node for the Newton AST is done by identifying which Newton node type that a host language program's node type maps to and then by calling the Newton API method `makeIrNodeSetValue` with the appropriate node type. For example a + node type in C would map to the Newton node type `kNewtonIrNodeType_Tplus`.

```
IrNode * plus = makeIrNodeSetValue(
  newton,
  kNewtonIrNodeType_Tplus,
  NULL,
  0
```

```
);
```

The third and the fourth parameters to `makeIrNodeSetValue` are the string value of the node and the numeric value of the node in case the nodes are strings or numbers.

Identifier nodes, however, require an additional treatment after making the nodes before they can be added to the Newton AST.

5.2.3 Building the Newton AST: Setting Physics Types of Identifiers

For the Newton parser, all of the signal scopes (see Section 5.1) define a Physics type. Every signal scope is noted by the Newton keyword "signal" and is preceded by an identifier which is the name of that Physics type. The Newton parser (written in C) stores the information about the signal scopes in a table of Physics struct's in memory. Now consider the Newton invariant scope shown below.

```
PendulumInvariant : invariant(a: acceleration@0, period: time) =
{
    a >= 2.2 * m / s ** 2,
    a <=  10 * m / s ** 2,
    period ~ (4 * Pi * Pi * pendulumLength / g) ** 0.5,
    period >= 3 * second,
    period <= 9 * second
}
```

When the Newton parser sees an identifier `a` associated with a Physics type `acceleration` in this scope, it creates a node with type `kNewtonIrNodeType_Tidentifier` with Physics type acceleration by looking it up in the table in memory (the Newton parser must have seen a base signal declaring `acceleration` before parsing this invariant scope).

Constructing a Newton identifier for the host language compiler is slightly different. Consider the C code below with language extension that allows using Physics types in Newton.

```
acceleration@0 foo = 4;
```

The host language compiler does not have the information about Physics types in memory like the Newton parser when it is translating its AST to a Newton AST (or building the Newton AST during the parsing phase). The host language compiler must call `newtonApiInit` with a file path to a Newton description to run the Newton lexer/parser

so that the Physics types are in memory and then call `newtonApiGetPhysicsTypeByName` on the string "acceleration" to ask the Newton API what Physics struct corresponds to the string "acceleration".

In general, having to create Newton identifier nodes for making Newton AST's will modify the type expression parsing step of the host language compiler as follows.

Algorithm 1: Parsing Type Expressions with Newton API

input : list of tokens

output : a type is set for the identifier node

type = next token in the list;

if *type is a data type in host language* **then**
  parse type as before.

else if *newtonApiGetPhysicsTypeByName(type) returns non-null* **then**
  set the type of the identifier related to this type in host IR as float;
  set the type of the identifier related to this type in Newton IR as the returned Physics struct;

else
  Unknown type. Add to compile errors;

end

Note that the above method assumes that the host language parser is building the Newton AST at the same time as it is building its own AST, but the same logic can be applied to the case where the host language compiler already built its AST and translating it to a Newton AST.

For languages that do not have static typing like Python, `newtonApiGetPhysicsTypeByNameAndSubindex` cannot be called at compile time. Therefore, the method should be called at runtime along with other Python type errors and syntax errors, just like how "'abc' + 8" throws an error at runtime. For example, in "myAcceleration = `read_from_accelerometer()`" written in Python, the type of the method `read_from_accelerometer` is not known at compile-time. For languages without static type checking, the steps just described above have to be performed at runtime. In addition, for languages like Python that do not have declaring type expressions in assignment statements such as in C, there needs to be an additional syntax that allows the

programmer to specify the type of variables such as

```
foobar = acceleration@0(7)
```

which would mean "foobar" is an acceleration object in the x axis with value of 7. This is similar to how objects are instantiated in Python, and it does not require significant changes from the original Python syntax except recognizing the sub-dimension notation "@". The algorithm then would apply to recognizing object instantiation expressions instead of type expressions, like the following.

Algorithm 2: Parsing Object Instantiation Expressions with Newton API

input : list of tokens

output : a type is set for the identifier node

type = next token in the list;

if *type is a name of a built-in type or defined type in host language* **then**
  parse type as before.

else if *newtonApiGetPhysicsTypeByName(type) returns non-null* **then**
  set the type of the identifier related to this type in host IR as a numeric type;
  set the type of the identifier related to this type in Newton IR as the returned Physics struct;

else
  Unknown type. Add to compile errors;

end

5.2.4 Building the Newton AST: Inserting a node into the Newton AST

Nodes in the Newton AST are inserted in depth-first manner. The Newton AST has the property that all nodes have only two child nodes, and the first vacant spot found is filled with the new node.

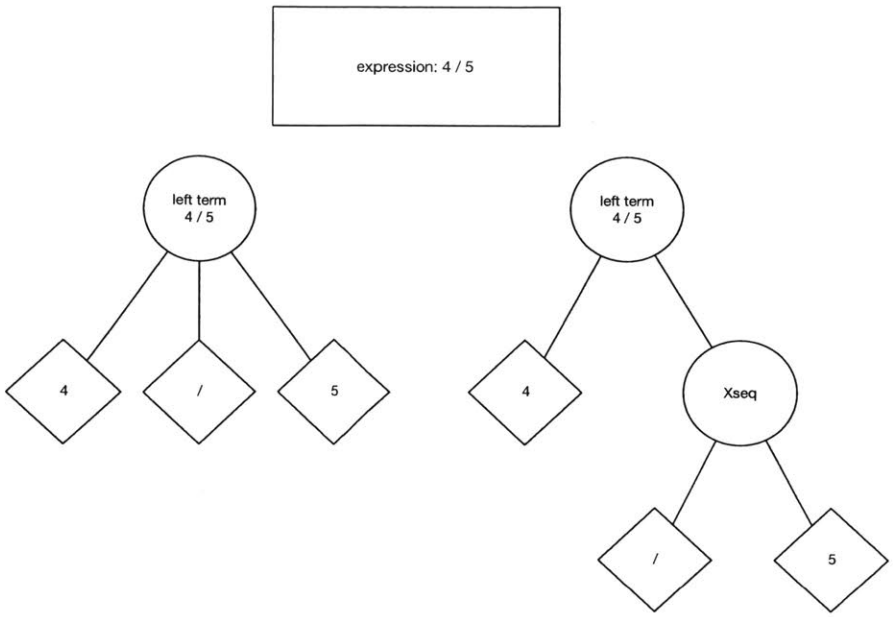


Figure 5-2: Comparison of AST's with and without a dummy Xseq for nodes with more than two children

Algorithm 3: Adding node without Xseq

input : a parent node, a node to be added

output : child is added to the tree

if *parent has a left child and a right child* **then**

    recurse on the right child;

    return;

end

if *parent has no child* **then**

    attach the input node to the parent as a left child;

    return;

end

if *parent has a left child* **then**

    attach the input node to the parent as a right child;

    return;

end

If a node has a type that should have more than two children, such as an expression having

two operands and a binary operation, then `newtonApiAddLeafWithChainingSeqNoLexer` method is used, which adds a dummy node of type Xseq before attaching a node to the tree.

Using an Xseq node has the advantage of simplifying tree traversals because all node types have the equal number of children. However, it is completely a matter of preference.

Algorithm 4: Adding node with Xseq

input : a parent node, a node to be added

output : child is added to the tree

if *parent has a left child and a right child* **then**

    recurse on the right child;

    return;

end

if *parent has no child* **then**

    attach to the parent as a left child;

    return;

end

if *parent has a left child* **then**

    attach Xseq to the parent as a right child;

    recurse on Xseq;

    return;

end

5.2.5 Newton AST Sample Tree

The following is a very simple Newton description and its corresponding Newton AST.

```
time : signal =
{
        name = "second" English
        symbol = "s";
        derivation = none;
}


Sample: invariant(t: time) =
{
    t < 1 * s
}
```

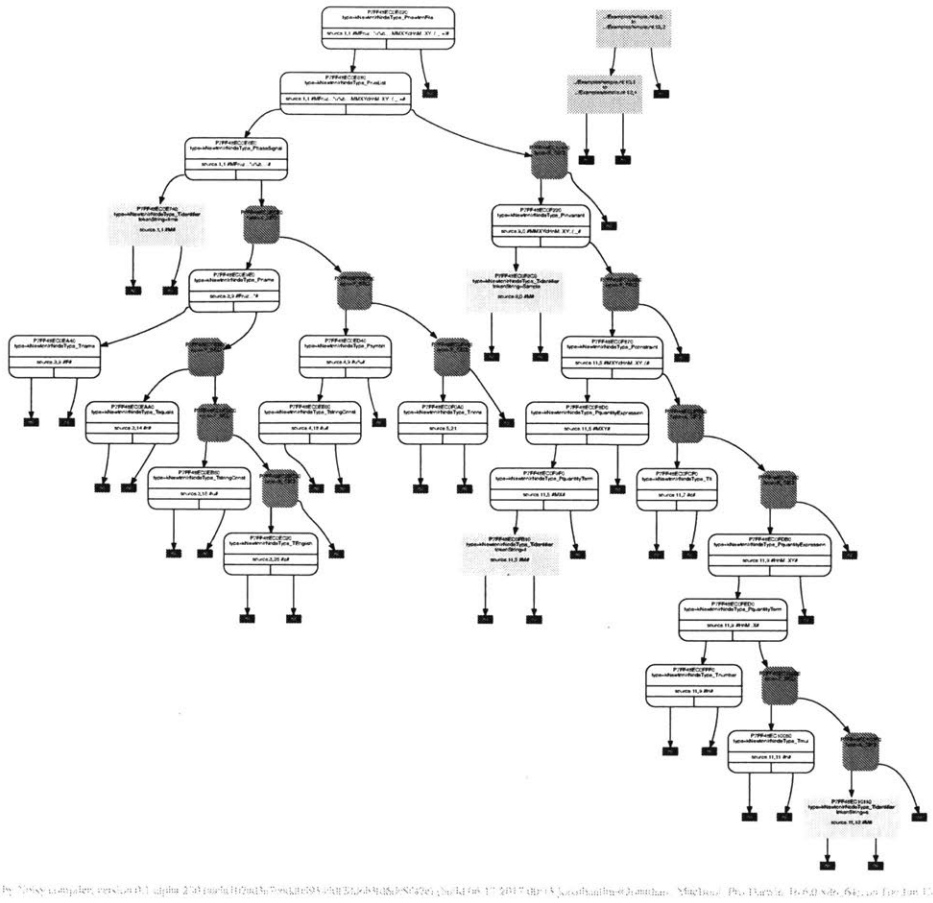


Figure 5-3: This picture has the Newton AST and the symbol tables of a simple Newton description. This image is automatically generated by one of the Newton compiler's backends.

The following is the Newton AST of the Newton description above. It may be necessary to zoom in to see the details more clearly.

The graph has been generated by the Dot visualization backend that is shared between the Newton compiler and the Noisy compiler, a compiler of another language. Using Dot backend is useful for visualizing the intermediate representation as well as for debugging any errors.

5.3 Newton AST Walk

The following two sections describe how to walk the Newton AST during the Newton API calls `newtonApiDimensionCheckTree` and `newtonApiSatisfiesConstraints`, respectively. The first method `newtonApiDimensionCheckTree` is used in contributing to compile-time errors for host language compilers as described in Section 6.1. The second method `newtonApiSatisfiesConstraints` is called at runtime after a host language compiler has transformed its intermediate representation to include the calls to `newtonApiSatisfiesConstraints`.

5.3.1 Newton AST Walk in newtonApiDimensionCheckTree: Determining Dimensional Consistency

The algorithm in this section describes a simple walk to perform type-checking of a binary operator expression subtree of a Newton AST according to the rules described below.

1. Binary operations multiply and divide can have two operands of different Physics types. The resulting expression will have a Physics type whose dimensions' exponents are the sum or the difference of the dimensions' exponents in the two operands. For example, in the following code

```
1        distance foo = 1;
2        time bar = 2;
3        velocity foobar = foo / bar;
4
```

   The variable `foo` has a Physics struct which contains the dimension `meter` of exponent 1 and the dimension `second` of exponent 0, and the variable `bar` has corresponding exponents set to 0 and 1. Then the exponents of variable `foobar` will be the exponents of `bar` subtracted from the exponents of `foo`.

2. The binary operation of exponentiation can also have two operands of different Physics types. The resulting expression will have a Physics type whose dimensions' exponents are the product of the exponents in the base expression node and the scalar value of the exponential expression node. The exponential expression node must have the

exponents of all dimensions equal 0. For example, the first code is allowed, but the second code isn't.

```
distance foo = 2;
area bar = foo ** 2;
```

,

```
distance foo = 2;
time bar = 3;
wrongType bar = foo ** bar;
```

3. Binary operations, add and subtract, must have two operands of same Physics types. The resulting expression will have a Physics type whose dimensions' exponents are the same as the exponents of either operands. Again with the same example, the first code is allowed, but the second code isn't.

```
distance foo = 1;
distance bar = 2;
distance foobar = foo + bar;
```

,

```
distance foo = 1;
time bar = 2;
wrongType foobar = foo + bar;
```

4. Compare operations of $>$, $<$, $>=$, $>=$, and $\sim$ follow the same rules as add and subtract.

5. The assign operation of $=$ follows the same rules as add and subtract.

The algorithm we describe next happens inside the compile-time Newton API call `newtonApiDimensionCheckTree` as it walks through the input Newton AST provided by a host language compiler. The symbol table of base and derived Physics types from the Newton API are used to look up the types of the leaf nodes, but the rest of this tree walk only involves the Newton AST that represents the host language program. This method is

used by host language compilers to build compile-time errors in Section 6.1. Type-checking of statements can be performed in a similar manner.

Algorithm 5: Tree Walk with Counting Children Nodes: Part 1 of 5

input : Newton AST representing an expression in a host language program

output : A compile-time error report

Procedure `checkExpression(`*currentNode*`)`

```
1   int expressionIndex = 1, termIndex = 0, factorIndex = 0;
2   int lowBinOpIndex = 0, midBinOpIndex = 0, highBinOpIndex = 0;
3   left = findNthNodeOfType(currentNode, termType, termIndex);
4     lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex, termIndex,
      factorIndex += checkTerm(left);
5   plusOrMinusNode = findNthNodeOfType(currentNode, plusOrMinusType,
      lowBinOpIndex);
6   while plusOrMinusNode is not null do
7       lowBinOpIndex ++;
8       right = findNthNodeOfType(currentNode, termType, termIndex);
9       lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex,
          termIndex, factorIndex += checkTerm(right);
10      check left term and right term have same Physics types. if not, add to compile
          errors and exit;
11      left = right;
12      plusOrMinusNode = findNthNodeOfType(currentNode, plusOrMinusType,
          lowBinOpIndex);
    end
13  return all the indices;
```

The algorithm is continued below.

Algorithm 6: Tree Walk with Counting Children Nodes: Part 2 of 5

input : Newton AST representing an expression in a host language program

output : A compile-time error report

Procedure `checkTerm(`*currentNode*`)`

```
1   int expressionIndex = 0, termIndex = 1, factorIndex = 0;
2   int lowBinOpIndex = 0, midBinOpIndex = 0, highBinOpIndex = 0;
3   left = findNthNodeOfType(currentNode, factorType, factorIndex);
4   lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex, termIndex,
      factorIndex += checkFactor(left);
5   initialize exponents of currentNode to be exponents of left;
6   mulOrDivNode = findNthNodeOfType(currentNode, mulOrDivType,
      midBinOpIndex);
7   while mulOrDivNode is not null do
8       midBinOpIndex++;
9       right = checkFactor(findNthNodeOfType(currentNode, factorType,
          factorIndex));
10      lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex,
          termIndex, factorIndex += checkFactor(right);
11      add or subtract exponents of right from termNode;
12      mulOrDivNode = findNthNodeOfType(currentNode, mulOrDivType,
          midBinOpIndex);
    end
13  return all the indices;
```

Continued.

Algorithm 7: Tree Walk with Counting Children Nodes: Part 3 of 4

input : Newton AST representing an expression in a host language program

output : A compile-time error report

Procedure `checkFactor(`*currentNode*`)`

```
1  int expressionIndex = 0, termIndex = 0, factorIndex = 1;
2  int lowBinOpIndex = 0, midBinOpIndex = 0, highBinOpIndex = 0;
3  expressionNode = findNthNodeOfType(currentNode, expressionType,
     expressionIndex);
4  if expressionNode is not null then
5      lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex,
         termIndex, factorIndex += checkExpression(expressionNode);
6      left = expressionNode;
7
   else
8      left = currentNode;
9
   end
10 exponentialNode = findNthNodeOfType(currentNode, exponentType,
     highBinOpIndex);
11 while exponentialNode is not null do
12     highBinOpIndex++;
13     right = findNthNodeOfType(currentNode, expressionType,
         expressionIndex);
14     lowBinOpIndex, midBinOpIndex, highBinOpIndex, expressionIndex,
         termIndex, factorIndex += checkExpression(right);
15     Check that right is dimensionless;
16     combine Physics types of left and right by multiplying or dividing exponents;
17     left = combined;
18     exponentialNode = findNthNodeOfType(currentNode, exponentType,
         highBinOpIndex);
   end
19 return all the indices;
```

The above methods walk through a given Newton AST performs checking on two operands. When performing a check with operands multiply or divide, it is necessary to add or subtract exponents of the base dimensions as mentioned above. When performing a check with operands add or subtract, no such operations are needed, but a check needs to be in place to make sure that left and right are of the same dimensional type. In the current implementation, this is achieved by having the Newton API keep track of a list of base dimensions per node, such as `distance`, `time`, `temperature`, and `electricalcharge`, and incrementing or decrementing those exponents as necessary.

Performing type-checking on factors requires multiplying or dividing the exponents of the base expression by the value of the exponential expression. We also ensure that the

exponents of the base dimensions in the exponential expression (if there is one) are all zeros.

Algorithm 8: Tree Walk with Counting Children Nodes: Part 4 of 4

input : Newton AST representing an expression in a host language program

output : A compile-time error report

Procedure `findNthNodeOfType(`*currentNode, targetType, index*`)`

```
1   if node->type is the targetType then
2       if index is 0 then
3           return this node;
        else
4           decrease index by 1;
        end
5
    end
6   targetNode;
7   if targetNode = findNthNodeOfType(currentNode->leftChild, targetType,
      index) is not null then
8       return targetNode;
9
    end
10  if targetNode = findNthNodeOfType(currentNode->rightChild, targetType,
      index) is not null then
11      return targetNode;
12
    end
13  return null;
```

Finally, `findNthNodeOfType` is a simple depth-first search of the given Newton AST that looks for the next node of a given type by keeping a counter of what we have seen so far in the given tree. This method essentially accomplishes what a parser would do with a stream of tokens to find out if the next token is in the first set of some type, but in this case, the input is an AST. The node passed to `findNthNodeOfType` and the

counter index represent the current position in the tree. A minor detail assumed about the input AST in the above algorithm is that kNewtonIrNodeType_quantityExpression, kNewtonIrNodeType_quantityTerm, and kNewtonIrNodeType_quantityFactor nodes are explicitly in the AST before the nodes whose types are subtypes of these three types are inserted into the AST. For example, before an identifier node is inserted into the Newton AST, kNewtonIrNodeType_quantityFactor is inserted first. Similarly, when the Newton parser encounters a parenthesis, kNewtonIrNodeType_quantityFactor is inserted before kNewtonIrNodeType_quantityExpression.

5.3.2 Newton AST Walk in newtonApiSatisfiesConstraints: Value Constraint Checking

The method newtonApiSatisfiesConstraints performs a Newton AST walk and is used at runtime as a result of the transformation described in Section 6.2. Note that this runtime library call happens at runtime after the transformation in Section 6.2 is performed at compile-time. In contrast to the method described in the previous section, which walks the host language program Newton AST at compile time, this method walks the Newton AST that represents the Newton description at runtime, except when looking up identifier leaf nodes in the parameter trees passed in from the host language runtime. Section 6.2 explains how the host language compiler selects variables from its program to form parameter trees and how the host language compiler intermediate representation is transformed.

The implementation for the invariant checking is essentially walking the Newton AST, setting values of the leaf nodes that correspond to the parameters passed in, and then propagating the values upward toward root of each constraint subtree. A constraint subtree is in the form LHS compareOp RHS. The values propagated to LHS and RHS are compared at the constraint subtree root, and if all of the constraints in the invariant passes, then the Newton API method newtonApiSatisfiesConstraints returns a NewtonAPIReport struct whose satisfiesValueConstraints field is set to true, and false otherwise.

Recall the first example of a Newton invariant in the beginning of last section. Each of the listed constraints, such as

```
a >= 3 * meter ** 2 / s
```

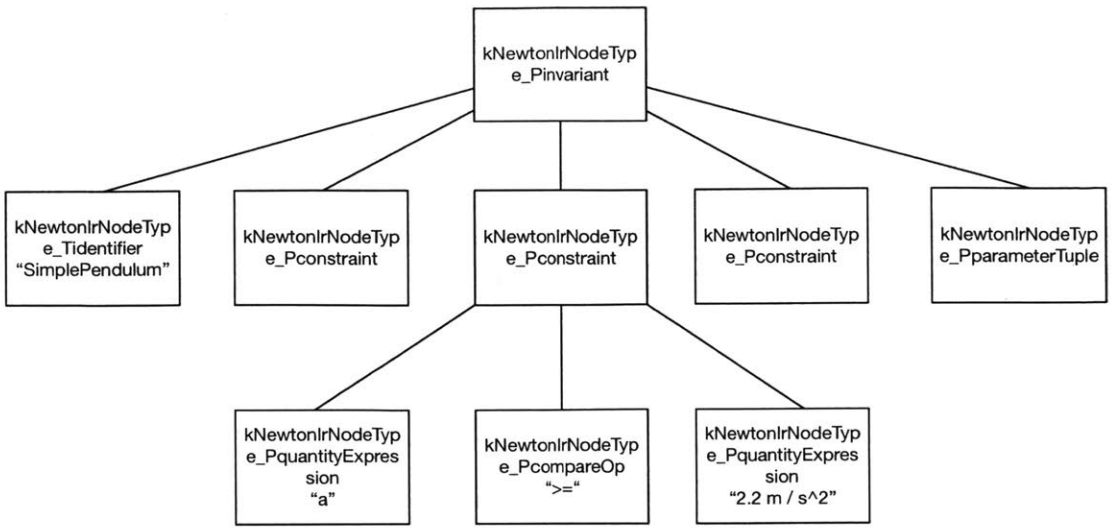


Figure 5-4: A sample invariant Newton subtree

are the nodes of type `kNewtonIrNodeType_Pconstraint`, whose children nodes are of types `kNewtonIrNodeType_PquantityExpression`, `kNewtonIrNodeType_PcompareOp`, and another `kNewtonIrNodeType_PquantityExpression` as we can see in Figure 5-4.

Walking each expression tree consists of finding all the factors that comprise each term and then applying appropriate binary operations between the two nodes at a time. After the values are propagated up to both expression tree roots of an invariant node, the Newton API compares the values in the two tree expression nodes.

Newton AST Walk in newtonApiSatisfiesConstraints: Runtime Dimensional Type Checking

Dimension constraints can fail at runtime if dimensions of the left expression and the right expression do not match. Because dimension type checking already happens at compile-time, runtime dimension checking is only necessary if runtime values would somehow result in different Physics types of the expressions inside the Newton description, which happens when a runtime numeric value is used inside an exponential expression of a base that has a Physics type. An exponential expression as a whole must be dimensionless, but parameters that go into it does not have to be. For example, suppose our invariant was

```
SimplePendulum : invariant(a: acceleration) =
{
    4 * m ** 2 ~ 4 * m ** (a / (2 * m / s ** 2)),
}
```

The invariant above is the same invariant as

```
SimplePendulum : invariant(a: acceleration) =
{
  a ~ 2 * m / s ** 2
}
```

The exponential expression itself must evaluate to be 2, or the dimensions of the LHS $4 * m^2$ and the RHS $4 * m^{a/(2*m/s^2)}$ wouldn't match. This example is the reason that dimensions are still checked at runtime because the values can change the dimension types if it is included in an exponential expression.

5.4 List of API Methods

These API methods are the interface between a host language compiler and Newton. The details of how two API methods `newtonApiDimensionCheckTree` and `newtonApiSatisfiesConstraints` are used and in what context are described in the next chapter 6 where we show how these two methods help host language program's compile-time dimensional type checking and runtime invariant checking.

The methods of the Newton compile-time library are listed below. For a complete description of the methods, see Appendix Section D.

Method Name	Returns	Parameter
newtonApiInit	struct State*	char* fileName
getPhysicsTypeByName	struct Physics*	struct State* newton, char* nameOfType
getPhysicsTypeByNameAndSubindex	struct Physics*	struct State* newton, char* nameOfType, int subindex
newtonApiDimensionCheckTree	struct ConstraintReport*	struct State* newton, struct IrNode* tree
newtonApiAddLeaf	void	struct State* newton, struct IrNode* parent, struct IrNode* newNode
newtonApiAddLeafWithChainingSeqNoLexer	void	struct State* newton, struct IrNode* parent, struct IrNode* newNode
newtonApiNumberParametersZeroToN	void	struct State* newton, struct IrNode* parameterTreeRoot

The Newton runtime library just consists of one method, shown below.

Method Name	Returns	Parameter
newtonApiSatisfiesConstraints	struct NewtonAPIReport*	struct State* newton, struct IrNode* parameterTreeRoot

5.5 Sample Newton Descriptions

In this section, we describe twelve real-world systems with Newton.

5.5.1 Pedometer Step Counter

The sensor platform used is a pedometer that uses a 3 axes accelerometer, a 3 axes gyro, and a processor. The application for the pedometer is a step counter which detects peaks in accelerometer data. The application that implements the algorithms is described in [7], and the transformation of the application code using the Newton API is described in Section A.1.

The base Physics types defined in the Newton description – `time`, `distance`, `speed`, and `angular_displacement` – only exist to be able to define the derived Physics types to be used as parameters of the invariant `accelerationAndGyro`. Note that it is not required to define the derived Physics types in terms of base Physics types: defining only `acceleration` and `angular_velocity` as base Physics types will not cause errors if they are treated as base Physics types by the host language program. However, having base Physics types makes the Newton description a more realistic representation of the system. There are 6 dimensions defined for many of the signals because there are 2 duplicate sensors of 3 degrees of freedom in acceleration and angular velocity.

One invariant of this system is the range of possible acceleration values, which is capped at $9.5m/s^2$, Usain Bolt's maximum recorded acceleration [11]. Another invariant of this system is that the tangential velocity is related to the angular velocity by the length of the arm if the sensors are worn at the wrist. Thus, the magnitude of the tangential velocity divided by the magnitude of the angular velocity or vice versa should equal the length of the arm. In this Newton description, the length of the human arm is broadly bounded between 0.1 meters and 2 meters. The third invariant shows sensor redundancy for the accelerometer and the gyro.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}
```

```
6
7  distance : signal(i: 0 to 5) = {
8      name = "meter" English
9      symbol = "m";
10     derivation = none;
11 }
12
13 speed : signal(i: 0 to 5) = {
14     derivation = distance@i / time;
15 }
16
17 acceleration : signal(i: 0 to 5) = {
18     derivation = speed@i / time;
19 }
20
21 angular_displacement : signal(i: 0 to 5) = {
22     name = "radian" English
23     symbol = "rad";
24     derivation = none;
25 }
26
27 angular_velocity : signal(i: 0 to 5) = {
28     derivation = angular_displacement / time;
29 }
30
31 SamplingTime : constant = 5 * 10 ** -4 * s;
32
33 maximumAcceleration: invariant(
34     x: acceleration@0,
35     y: acceleration@1,
36     z: acceleration@2
37 ) = {
38     x < 9.5 * m / s ** 2,
39     x > - 9.5 * m / s ** 2,
40     y < 9.5 * m / s ** 2,
41     y > - 9.5 * m / s ** 2,
42     z < 9.5 * m / s ** 2,
43     z > - 9.5 * m / s ** 2
44 }
45
46 accelerationAndGyro : invariant(
47     x: acceleration@0,
48     y: acceleration@1,
49     z: acceleration@2,
50     row: angular_velocity@0,
51     pitch: angular_velocity@1,
52     yaw: angular_velocity@2
53 ) = {
54   # rectangular integration estimation
55     ((x * SamplingTime) ** 2 + (y * SamplingTime) ** 2 + (z * SamplingTime)
       ** 2) / (row ** 2 + pitch ** 2 + yaw ** 2) < (2 * m / rad) ** 2,
56     (row ** 2 + pitch ** 2 + yaw ** 2) / ((x * SamplingTime) ** 2 + (y *
       SamplingTime) ** 2 + (z * SamplingTime) ** 2) > (0.1 * rad / m) ** 2
57 }
58
59 accelerationRedundancy: invariant (
```

```
    x1: acceleration@0,
    y1: acceleration@1,
    z1: acceleration@2,
    x2: acceleration@3,
    y2: acceleration@4,
    z2: acceleration@5
) = {
    x1 ~ x2,
    y1 ~ y2,
    z1 ~ z2
}

gyroRedundancy: invariant (
    x1: angular_velocity@0,
    y1: angular_velocity@1,
    z1: angular_velocity@2,
    x2: angular_velocity@3,
    y2: angular_velocity@4,
    z2: angular_velocity@5
) = {
    x1 ~ x2,
    y1 ~ y2,
    z1 ~ z2
}
```

5.5.2 Activity Classifier

The goal of this system is to collect sensor data of a smart-watch worn by a person doing daily activities and to be able to label activities on each row of sensor data. Sensors used by the activity classifier are accelerometers in three axes, gyro in three axes, pressure sensor, and magnetic field sensor in three axes. In addition to the sensors, the smart-watch would have a processor, a storage, and a network driver to be able transmit the sensor data to another computer.

The first invariant used in this system is identical as in the pedometer step counter example. The second invariant encodes an acceleration sensor redundancy.

While recording sensor data, if the invariants are violated, the smart-watch could log errors to a file.

```
time : signal = {
    name = "time" English
    symbol = "s";
    derivation = none;
}

distance : signal(i: 0 to 5) = {
    name = "meter" English
```

```
    symbol = "m";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

speed : signal(i: 0 to 5) = {
    derivation = distance@i / time;
}

acceleration : signal(i: 0 to 5) = {
    derivation = speed@i / time;
}

angular_displacement : signal(i: 0 to 2) = {
    name = "radian" English
    symbol = "rad";
    derivation = none;
}

angular_velocity : signal(i: 0 to 2) = {
    derivation = angular_displacement@i / time / 60;
}

magnetic_field : signal = {
    name = "Tesla" English
    symbol = "T";
    derivation = none; # not a base SI unit, but can be defined as a Newton
    base unit
}

TimeBetweenSensors: constant = 5 * 10 ** -6 * s;
SamplingTime : constant = 5 * 10 ** -4 * s;

accelerationAndGyro : invariant(
    x: acceleration@0,
    y: acceleration@1,
    z: acceleration@2,
    row: angular_velocity@0,
    pitch: angular_velocity@1,
    yaw: angular_velocity@2
) = {
  # rectangular integration estimation
    ((x * SamplingTime) ** 2 + (y * SamplingTime) ** 2 + (z * SamplingTime)
    ** 2) / (row ** 2 + pitch ** 2 + yaw ** 2) < (2 * m / rad) ** 2,
    (row ** 2 + pitch ** 2 + yaw ** 2) / ((x * SamplingTime) ** 2 + (y *
    SamplingTime) ** 2 + (z * SamplingTime) ** 2) > (0.1 * rad / m) ** 2
}

redundantAccelerometers : invariant(
    x1: acceleration@0,
    y1: acceleration@1,
```

```
    z1: acceleration@2,
    x2: acceleration@3,
    y2: acceleration@4,
    z2: acceleration@5
) = {
    x1 ~ x2,
    y1 ~ y2,
    z1 ~ z2
}
```

5.5.3 Maintaining Vehicle Distance

This system is about maintaining a proper distance between a row of autonomous vehicles, and it aims at stabilizing distances between the vehicles as well as quickly responding to any disturbances while avoiding collisions between the vehicles [9].

Sensors used by each vehicle are accelerometers in three axes, gear shaft rotation sensor, and a sonar radar sensor (gives distance from the previous vehicle). Each vehicle would also carry a processor, a storage, and a network driver that allows the vehicle to communicate with a remote server or with other vehicles. The Newton description specifies invariants for sensors in each vehicle in order for this system to behave correctly, namely to maintain a reference distance and a certain relative velocity from the previous vehicle.

```
time : signal = {
    name = "time" English
    symbol = "s";
    derivation = none;
}

distance : signal = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

speed : signal = {
    derivation = distance / time;
}

acceleration : signal = {
        derivation = speed / time;
}

angular_displacement : signal = {
        name = "radian" English
        symbol = "rad";
        derivation = none;
```

```
}

angular_velocity : signal = {
        derivation = angular_displacement / time / 60;
}

TimeBetweenSensors: constant = 5 * 10 ** -6 * s;
ReferenceDistance: constant = 1.3 * m;

keepDistance: invariant(
    x: distance
) = {
    x > ReferenceDistance
}

velocityBound: invariant(
    distance_to_prev_car: distance,
    current_speed: speed
) = {
    distance_to_prev_car - ReferenceDistance > current_speed *
    TimeBetweenSensors
}
```

5.5.4 Weather Balloon

This system describes a weather balloon that collects atmospheric data. Sensors on a typical weather balloon detect temperature, altitude, pressure, humidity, and more, but this Newton description focuses on temperature and pressure sensors. The balloon would also contain a processor and a storage device that contains recorded sensor information as well as any error logs.

The invariants are the properties of the International Standard Atmosphere [10]. In the ISA troposphere layer, the altitude, the air pressure, and the atmospheric temperature are related to each other by the gas laws. If the invariants are violated, the software of the device recording sensor data executes an error handler function that might do something like recording to an error log.

```
time : signal = {
    name = "time" English
    symbol = "s";
    derivation = none;
}

distance: signal = {
    name = "meter" English
    symbol = "m";
    derivation = none;
```

```
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

temperature : signal = {
    name = "Kelvin" English
    symbol = "K";
    derivation = none;
}

area : signal = {
    derivation = distance ** 2;
}

speed : signal = {
    derivation = distance / time;
}

acceleration : signal = {
    derivation = speed / time;
}

force : signal =  {
    name = "Newton" English
    symbol = "N" ;
    derivation = mass * acceleration;
}

pressure: signal =  {
    name = "Pascal" English
    symbol = "Pa" ;
    derivation = force / area;
}

groundpressure: constant = 101325 * Pa;
groundtemp: constant = 288.15 * K;

# http://home.anadolu.edu.tr/~mcavcar/common/ISAweb.pdf
altitudeAndPressureTroposphere: invariant(
    altitude: distance,
    airpressure: pressure
) = {
    airpressure > groundpressure * (1 - 0.0065 * (altitude * K) / (groundtemp
     * m)) ** 5.2561 - 200 * Pa,
    airpressure < groundpressure * (1 - 0.0065 * (altitude * K) / (groundtemp
     * m)) ** 5.2561 + 200 * Pa
}

# http://home.anadolu.edu.tr/~mcavcar/common/ISAweb.pdf
altitudeAndTemperatureTroposphere: invariant(
    altitude: distance,
    airtemp: temperature
```

```
) = {
    airtemp > groundtemp - 6.5 * K * altitude / (1000 * m) - 8 * K, # 8 * K
    == error tolerance
    airtemp < groundtemp - 6.5 * K * altitude / (1000 * m) + 8 * K
}
```

5.5.5 GPS Walking

This is example describes GPS-Walking from [15], where a person is walking with a smart-watch containing a GPS, an accelerometer, a gyro, and a pressure sensor. The application simply alerts the person to move faster if the speed detected is below a target speed. The smart-watch, in addition to the sensors, contains a processor, a storage medium, and a network driver. The invariant is that the difference in GPS location should roughly equal the velocity of the person multiplied by the sampling time. There are 6 dimensions defined for many of the signals because there are 2 duplicate sensors of 3 degrees of freedom in distance and velocity.

When invariants are violated, the application may do something like alerting the person wearing the smart-watch that either GPS or accelerometer may need to be re-calibrated.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

distance: signal(i: 0 to 5) =  {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

speed : signal(i: 0 to 5) = {
    derivation = distance@i / time;
}

acceleration : signal(i: 0 to 5) = {
    derivation = speed / time;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

SamplingTime: constant = 5 * 10 ** -5 * s;
```

```
28
29 GPSAndAccelerometerMatch : invariant (
30     ds: distance@3,
31     v: speed
32 ) = {
33     # double rectangular integration estimation
34     ds > v * SamplingTime - 1 * m,
35     ds < v * SamplingTime + 1 * m
36 }
37
38 GPSRedundancy: invariant(
39     x1: distance@0,
40     y1: distance@1,
41     x2: distance@3,
42     y2: distance@4
43 ) = {
44     x1 ~ x2,
45     y1 ~ y2
46 }
```

5.5.6 Sensor Life

This example is a simulation of Conway's Game of Life except that the cells "turn on" if their temperatures reach a certain temperature threshold and "turn off" if their temperatures fall below the threshold. Each cell is a board with temperature sensors, a processor, a memory, a cooling unit, and a heating unit. Sensors used for each cell are 9 temperature sensors, 8 to sense the temperatures of their neighbors and 1 to sense its own temperature. The rules for the Game of Life are reproduced here below [3]:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

This example differs from the Conway's Game of Life in the initialization step. Cells with temperature higher than a threshold are initialized as alive. When a cell receives a signal to kill itself because it didn't meet the Conway rules for survival, it will cool itself to

below the threshold temperature. Likewise, when a cell receives a signal to be brought back alive, it will heat itself to above the threshold temperature.

The invariants are simply that each cell has a maximum tolerable temperature and that each of the 9 temperature sensors has a duplicate.

```
temperature: signal(i: 0 to 17) = {
    name = "Celsius" English
    symbol = "C";
    derivation = none;
}

maxTemp: constant = 120 * C;

tempInRange: invariant(
    neighborTempNW: temperature@0,
    neighborTempN: temperature@1,
    neighborTempNE: temperature@2,
    neighborTempE: temperature@3,
    neighborTempW: temperature@4,
    neighborTempSW: temperature@5,
    neighborTempS: temperature@6,
    neighborTempSE: temperature@7,
    selfTemp: temperature@8
) = {
    neighborTempNW < maxTemp,
    neighborTempN < maxTemp,
    neighborTempNE < maxTemp,
    neighborTempE < maxTemp,
    neighborTempW < maxTemp,
    neighborTempSW < maxTemp,
    neighborTempS < maxTemp,
    neighborTempSE < maxTemp,
    selfTemp < maxTemp
}

temperatureRedundancy: invariant(
    neighborTempNW1: temperature@0,
    neighborTempN1: temperature@1,
    neighborTempNE1: temperature@2,
    neighborTempE1: temperature@3,
    neighborTempW1: temperature@4,
    neighborTempSW1: temperature@5,
    neighborTempS1: temperature@6,
    neighborTempSE1: temperature@7,
    selfTemp1: temperature@8,
    neighborTempNW2: temperature@9,
    neighborTempN2: temperature@10,
    neighborTempNE2: temperature@11,
    neighborTempE2: temperature@12,
    neighborTempW2: temperature@13,
    neighborTempSW2: temperature@14,
    neighborTempS2: temperature@15,
    neighborTempSE2: temperature@16,
    selfTemp2: temperature@17
```

```
) = {
    neighborTempNW1 ~ neighborTempNW2,
    neighborTempN1 ~ neighborTempN2,
    neighborTempNE1 ~ neighborTempNE2,
    neighborTempE1 ~ neighborTempE2,
    neighborTempW1 ~ neighborTempW2,
    neighborTempSW1 ~ neighborTempSW2,
    neighborTempS1 ~ neighborTempS2,
    neighborTempSE1 ~ neighborTempSE2,
    selfTemp1 ~ selfTemp2
}
```

5.5.7 Ball Dropped from a Height

This application is a simple Physics experiment recording gravitational potential energy and kinetic energy of a bouncing ball. The ball contains a processor, a storage, a board with 3 axes accelerometers and a sensitive altitude sensor. The assumption is that the altitude sensor can detect the change in height of the bouncing ball. If the altitude sensor is not accurate enough, a sonar radar located at the bottom of the ball could also measure the distance between the ball and the ground. Both altitude sensor and the sonar radar return the height of the ball.

This is an example of not having a hardware manufacturer involved because the programmer sets up the experiment with sensors embedded in a ball and runs software on it. Thus, the programmer can specify whatever invariant that is appropriate for the experiment. The ball contains a processor and a storage that together read data from the sensors and record them to a file.

The invariant here is that every time the ball hits the ground, it should lose some energy, so the sum of the mechanical energy of the ball should be less than the initial gravitational potential energy.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}
```

```
12
13 distance : signal(i: 0 to 2) = {
14     name = "meter" English
15     symbol = "m";
16     derivation = none;
17 }
18
19 speed : signal(i: 0 to 2) = {
20     derivation = distance / time;
21 }
22
23 acceleration : signal(i: 0 to 2) = {
24     derivation = speed / time;
25 }
26
27 force : signal(i: 0 to 2) = {
28     name = "Newton" English
29     symbol = "N" ;
30     derivation = mass * acceleration@i;
31 }
32
33 energy: signal(i: 0 to 2) = {
34     name = "Joule" English
35     symbol = "J";
36     derivation = force@i * distance;
37 }
38
39 g : constant = 9.8 * m * s ** -2;
40 SamplingTime: constant = 5 * 10 ** -3 * s;
41 initialHeight: constant = 10 * m;
42 myMass: constant = 1 * kg;
43
44 mechanicalEnergyDecreasing: invariant (
45     h: distance@2,
46     x: acceleration@0,
47     y: acceleration@1,
48     z: acceleration@2
49 ) = {
50     myMass * g * initialHeight / SamplingTime >= myMass * g * h /
      SamplingTime + 0.5 * myMass * (x ** 2 + y ** 2 + z ** 2) * s - 10 * J /
      SamplingTime
51 }
```

5.5.8 Jet Engine

This application is a software running on top of a jet engine. Sensors used are the mass flow rate sensor and a pressure sensor, and there is a small computer that reads those sensor values and constantly monitors the state of the engine. The invariant is the Moore-Greitzer Jet Engine Model as described in [12]. The invariant should be preserved while the jet engine is performing other tasks such as taking inputs from the pilot.

When the invariant is violated, the application can do something like alerting the pilot on the display or recording to an error log.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

mass_flow_rate: signal = {
    derivation = mass / time;
}

distance : signal = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

area : signal = {
    derivation = distance ** 2;
}

speed : signal = {
    derivation = distance / time;
}

acceleration : signal = {
    derivation = speed / time;
}

force : signal = {
    name = "Newton" English
    symbol = "N" ;
    derivation = mass * acceleration;
}

pressure: signal = {
    name = "Pascal" English
    symbol = "Pa" ;
    derivation = force / area;
}

SamplingTime: constant = 5 * 10 ** -3 * s;

MooreGreitzerJetEngineModel: invariant(
    x: mass_flow_rate,
    y: pressure
) =
```

```
{
    x * kg ** 2 * s ** 4 > -y * kg ** 3 * s ** 3 / Pa - 1.5 * (x *
    SamplingTime) ** 2 * kg * s ** 3  - 0.5 * (x * SamplingTime) ** 3 * s ** 3
     - 0.5 * kg ** 3 * s ** 3,
    y / SamplingTime * kg ~ 3 * x * Pa - y * kg / s
}
```

5.5.9 Reactor Rod Cooling

The application is software that monitors the state of a reactor. The reactor has sensors, a processor, a storage device, and a display for the human operator. Sensors used are two temperature sensors, one observing the surface temperature of the rod and the other measuring the temperature of the water used to cool the reactor rod. The invariant is based on the Newton's Law of Cooling. Based on the rate of cooling possible observed in the beginning and the maximum amount of time the reactor is allowed to cool, detects if the cooling can take place within the maximum allocated amount of time. If the invariant is violated, then the human operator would be alerted.

```
time : signal =
{
    name = "second" English
    symbol = "s";
    derivation = none;
}

temperature : signal(i: 0 to 1) =
{
    name = "Celsius" English
    symbol = "C";
    derivation = none;
}

maxCoolingTime: constant = 50000 * s;
NewtonsLawConstant: constant = 0.5 / s;
targetCoolingTemperature: constant = 70 * C;

# if we can't cool in max amount of time, something has gone wrong.
RodCooling: invariant(
    rodTemp: temperature@0,
    waterTemp: temperature@1
) =
{
    rodTemp - NewtonsLawConstant * (rodTemp - waterTemp) * maxCoolingTime <=
    targetCoolingTemperature
}
```

5.5.10 Airplane Altitude and Speed

The application is an autopilot that monitors the state of the aircraft as well as fly the aircraft. Sensors used are two pressure sensors, one in an altimeter and the other in the pitot tube. There is also a processor, a storage, and a display for the pilot.

The invariant is that the pressure measured in the altimeter should match the pressure reading in the pitot tube. The two sensor readings can differ because the pitot tube can have a tape blocking it or some dust may get stuck inside. If the two pressure readings are contradictory, then the pilot will get erroneous altitude readings and velocity readings, which may critically endanger the safety of the entire aircraft. There was an accident that occurred due to the two pressure reading inconsistencies where the pilots were unaware of the true altitude and the velocity that the aircraft should be going [1].

When the invariant is violated, the autopilot would simply alert the pilot on the display that the sensor readings are inconsistent.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

distance : signal = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

area : signal = {
    derivation = distance ** 2;
}

speed : signal = {
    derivation = distance / time;
}

acceleration : signal = {
    derivation = speed / time;
}

force : signal =  {
```

```
    name = "Newton" English
    symbol = "N" ;
    derivation = mass * acceleration;
}

# 0 is the static pressure sensor for altimeter
# 1 is the static pressure sensor for pitot tube
pressure: signal(i: 0 to 1) = {
    name = "Pascal" English
    symbol = "Pa" ;
    derivation = force / area;
}

AltimeterPitotPressuresShouldMatch: invariant(
    altimeter_pressure: pressure@0,
    pitot_pressure: pressure@1
) = {
    altimeter_pressure > pitot_pressure - 10 * Pa,
    altimeter_pressure < pitot_pressure + 10 * Pa
}
```

5.5.11 Motorized Wheel Chair

The application is the software running a wheel chair, which contains various sensors, a processor, a storage, a display for the passenger, and a motor. Sensors that we focus on in this Newton description are 3 axes accelerometer, a pressure sensor in the seat, and a temperature sensor.

The invariant is that when the wheel chair is turned on, there must be a passenger sitting in the chair. That means the pressure sensor should be able to detect the passenger and that the passenger's body temperature must be of a human being. In addition, there is a speed limit to the wheel chair's movement.

When the invariant is violated, the wheel chair can alert through the display or slow down to a stop for safety.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

temperature: signal = {
    name = "Celsius" English
    symbol = "C";
    derivation = none;
}

```

```
mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

distance : signal(i: 0 to 2) = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

area : signal = {
    derivation = distance ** 2;
}

speed : signal(i: 0 to 2) = {
    derivation = distance / time;
}

acceleration : signal(i: 0 to 2) = {
    derivation = speed / time;
}

force : signal = {
    name = "Newton" English
    symbol = "N" ;
    derivation = mass * acceleration;
}

pressure: signal = {
    name = "Pascal" English
    symbol = "Pa" ;
    derivation = force / area;
}

SamplingTime: constant = 5 * 10 ** -3 * s;

SafetyCheck: invariant(
    seatPressure: pressure,
    passengerTemp: temperature
) = {
    seatPressure > 0 * Pa,
    passengerTemp > 34 * C,
    passengerTemp < 39 * C
}

SpeedCheck: invariant(
    x: acceleration@0,
    y: acceleration@1,
    z: acceleration@2
) = {
    x + y + z < 5 * m / s ** 2
}
```

5.5.12 Car Tire Pressure and Acceleration Range

The application is the software monitoring the state of a car. The car would have various types of sensors and a small computer with a processor, a storage, and a display. Sensors discussed in this example are the pressure sensor for a car's tires and the accelerometer. The invariant is simply that tire pressures of the car cannot get too low or too high.

When the invariant is violated, there can be an alert on the display for the driver.

```
time : signal = {
    name = "second" English
    symbol = "s";
    derivation = none;
}

mass : signal = {
    name = "kilogram" English
    symbol = "kg";
    derivation = none;
}

distance : signal = {
    name = "meter" English
    symbol = "m";
    derivation = none;
}

area : signal = {
    derivation = distance ** 2;
}

speed : signal = {
    derivation = distance / time;
}

acceleration : signal = {
    derivation = speed / time;
}

force : signal = {
    name = "Newton" English
    symbol = "N" ;
    derivation = mass * acceleration;
}

pressure: signal = {
    name = "Pascal" English
    symbol = "Pa" ;
    derivation = force / area;
}

TirePressureRange: invariant(
    tire_pressure: pressure
```

```
45 ) = {
46     tire_pressure > 206843 * Pa,
47     tire_pressure < 241317 * Pa
48 }
```

Chapter 6

Applications

This chapter shows useful applications of Newton at compile-time and at runtime of a host language program. A compiler that would perform the applications mentioned in this chapter is not implemented as part of this thesis. The compile-time application uses the Newton compile-time library call `newtonApiDimensionCheckTree` to contribute to host language compiler error messages about Physics types. The runtime usage of the Newton runtime library call `newtonApiSatisfiesConstraints` checks whether an invariant is satisfied given a list of host language variables as seen in Section 5.3.2. This chapter explores two compile-time transformations to a host language program that takes advantage of invariants written in a Newton description – the first transformation trades off performance for reliability by checking invariants with host language variable values at runtime, and the second transformation trades off reliability for performance by exploiting equivalence relations in Newton invariants in order to reduce sensor redundancy logic in the host language program at compile-time.

6.1 Compile-Time Checks

The compile-time checks ensure that the program in a host language is dimensionally consistent, which means the operands of expressions and statements have valid Physics types. Section 5.3.1 described how the Newton library method `newtonApiDimensionCheckTree` performs dimensional type checking on an expression or a statement Newton subtree and

how a host language compiler can use the method to generate compile-time errors about dimensional types throughout its program. Now that we have the tools to check statements and expressions, here is the algorithm that performs compile-time dimension checks on statements and expressions using the Newton compile-time library, given any host language

program.

Algorithm 9: Algorithm for Dimensional Type Checking on Host Language Program Expressions and Statements

```
input  : a Newton AST representing a host language program with Physics types not
         filled in yet
output : A compile-time error report
Initialize an error report struct;
Recurse left child if exists;
Recurse right child if exists;
if current node is an unknown token in a type declaration then
    Physics* type = newtonApiGetPhysicsTypeByNameAndSubindex(...);
    Find the identifier related to this type declaration;
    Store Physics type in the type field in the Newton node of this identifier;
    Update the type field of this identifier in a symbol table;
    Mark the current node as "NewtonTyped";
end
if current node is a usage of an identifier then
    if type in the symbol table is a Physics type then
        Mark the current node as "NewtonTyped";
    end
end
if the left child or the right child is marked "NewtonTyped" then
    Mark the current node as "NewtonTyped";
end
if the current node is marked "NewtonTyped" and is an expression or a statement then
    Call newtonApiDimensionCheckTree by passing in the current node;
    Add any errors to compile-time errors;
end
```

The input to this algorithm is the Newton AST of a program written in a host programming language. The above algorithm is called on all expression and

statement subtrees in the Newton AST. Whenever the host language compiler sees an unknown token in a type declaration, it will call the Newton library method `newtonApiGetPhysicsTypeByNameAndSubindex` to find out its Physics type. The host language compiler can then mark any expression subtree in its AST that contains a Newton Physics type variable to be a candidate and pass into another Newton library method `newtonApiDimensionCheckTree`, which will return information about whether the host language expression is dimensionally consistent. If the Newton API call indicates that an expression has a type dimension mismatch, then the host language compiler can add the warning in the result of the method `newtonApiDimensionCheckTree` to its own list of compile warnings.

Figure 6-1 shows a simple example code "distance foo = 4; time bar = 3; speed baz = 2; speed foobar = foo / bar + baz;". The identifiers are foo, bar, baz, and foobar. In the type expression "distance foo", the token "distance" is unknown, so the host language compiler makes the Newton API call `newtonApiGetPhysicsTypeByNameAndSubindex` to find out what Physics "distance" is. When this Newton API call happens, the compiler marks the current Newton AST node. To find out if the current subtree contains any node that is marked, the algorithm recursively checks if the left and the right child nodes are marked. At the end of this algorithm, all expressions that have any leaf nodes that are marked will also be marked. In the marked subtrees, only the leaf nodes that are marked will be the identifier nodes that have Physics types defined in Newton. This fact can be used to find variables that have Newton types in the AST in Section 5.3.2.

6.1.1 Type Inference

The Newton compile-time library supports a simple type inference on an expression or a statement given a Newton AST. The type inference rule is that if a node does not have any Physics type and is a numeric type, it will remain a wildcard Physics type unless a sibling node in the AST has a Physics type, in which case the wildcard Physics type will infer the Physics type from the sibling node. For example, "distance foo = 5;" does not violate type checking rules because the left hand side will have the type distance, and the right hand

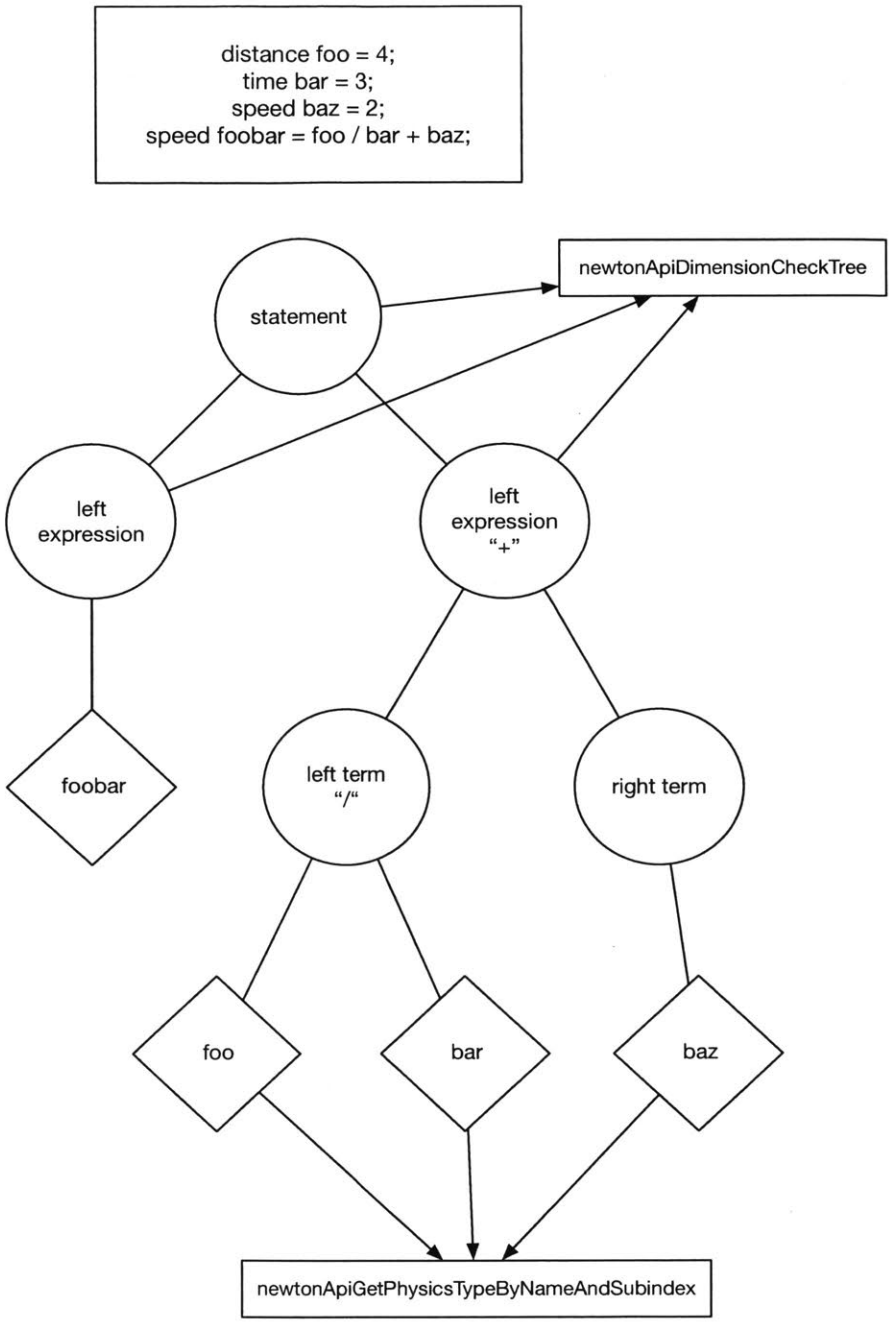


Figure 6-1: Dimensional type checking is performed in a bottom-up approach. Given a Newton statement tree, the host language compiler needs to call `newtonApiGetPhysicsTypeByNameAndSubindex` to find out the Physics types of the leaf identifier nodes. Once the leaf identifier nodes' Physics types are set, the host language compiler can check that expressions and statements built from those nodes have operands with valid Physics types by calling `newtonApiDimensionCheckTree`.

side will have the wildcard Physics type where both sides are the sibling nodes of an assign statement. The right hand side 5 will assume the type distance. However, in "time bar = 2; distance foo = 2 + bar;" would not pass dimensional checks because the right hand side, after 2 assumes the type time, would have the Physics type time where as the left hand side would have distance. Because an assign statement requires the type of left hand side and the right hand side to be the same, the dimensional type-checking would return a compile-time error.

This way of doing type inference is similar to duck-typing in that for a numeric type node we try a Physics type that works and if it doesn't fit within the context, the Newton API reports a compile-error. We try a binary operation on the wildcard Physics type by assigning it a Physics type inferred from a neighboring node, and if there is no error, the numeric type becomes that Physics type. That is, if the numeric type behaves like a Physics type, then it must be that Physics type [4]. The difference of this method from duck-typing is that the type inference is not based on the type of the node itself but on the types of the neighboring nodes. Duck-typing often occurs on dynamically typed languages like Python, but this method happens at compile-time when `newtonApiDimensionCheckTree` is called.

Performing this type inference has an advantage that the syntax in the host language using Newton becomes simple. A main purpose of Newton is to reduce the programmer's burden by describing assumptions about hardware sensor values. This means enabling the programmer to develop software without worrying about too much additional syntax. Without this type inference, the programmer would have to write "time bar = 2 * s; distance foo = 2 * s + bar;". While this syntax is reasonable, the programmer needs to know additional information about the programming syntax of how to incorporate these units into every expression and statements. In addition, the programmer need to know decide what the units are from the Newton description, in which case the Newton description is no longer hidden from the programmer's point of view. The host language compiler would also need to reserve additional strings that appear inside Newton base signal scopes as reserved words. The disadvantage of using this type inference method is that "distance foo = 5;" might not appear dimensionally consistent to the programmer, but the approach taken in this thesis is that the overall burden on the programmer will be reduced by this type inference.

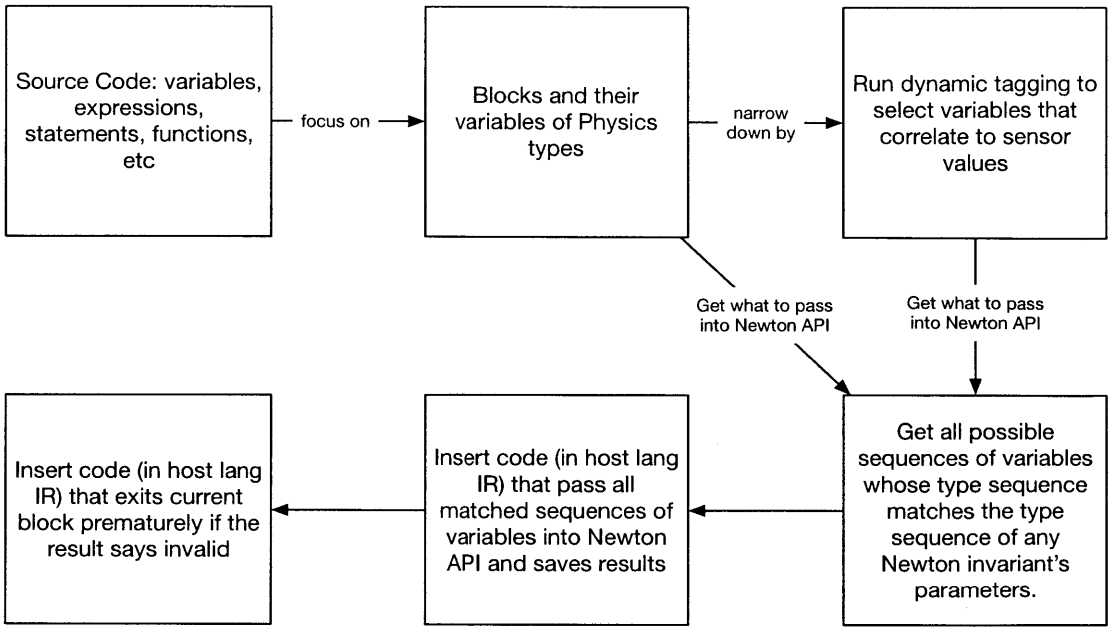


Figure 6-2: Overview of how a host language compiler interacts with Newton in the premature exit transformation

6.2 Transformation To Check Invariants

The next two sections explain how to transform the host language compiler's AST at compile-time in order to make the host language program more robust or improve its performance. This thesis introduces two transformations, one that changes a host language program's code path when values of variables violate any Newton invariants at runtime and another that simplifies code by exploiting sensor redundancy encoded in Newton invariants.

One transformation that a host language compiler can make is inserting the call to `newtonApiSatisfiesConstraints` with appropriately selected variables and execute an error handler function when an invariant in the Newton description is found to be violated by the host language program at runtime. We demonstrate that reasonable and useful transformations of the source code can happen at the level of a block, which is a function scope or a loop scope. See Figure 6-2 for an overview of this section. To see a sample code that may benefit from this transformation, see Section A.1 or the step-by-step example from the end of Section 4.

The idea behind this transformation is that calling the Newton runtime library essentially

checks the assumptions of a host language program about its sensor signal values. If those assumptions about low-level hardware sensor values are violated, then the high-level application logic may not be valid. The approach of this transformation is that executing an error handler instead of the current code allows a system to fail faster and thus make the system safer [6]. The host language must create some syntax that allows the programmer to write error handling code when invariant is assumed to have failed, similar to how exceptions in Java are thrown and caught.

A core benefit of the Newton API is that the programmer does not need to know the exact contents of the Newton description. The advantage of programming without knowing the contents of the Newton description is reducing knowledge burden on the programmer so that the programmer can focus on developing software. In case the programmer wants more information, the programmer still has access to look at the Newton description file provided by the hardware manufacturer on a sensor platform.

The explicit benefits to the programmer are the following:

1. The programmer does not have to say which set of variables can bind to which invariants in the Newton description file.

2. The programmer does not have to know the names of the variables as defined in the Newton description. For example, in the invariant of the previous section, the programmer does not have to specify that a parameter corresponds to a variable named a but just needs to set the type to $acceleration$. This feature is useful if more than one $acceleration$ type variables are passed into the Newton API.

3. Given two or more sets of variables of identical Physics types, the programmer does not need to know which of the sets should be passed into the Newton API.

Note that at runtime the binary of the source code needs to be able to call Newton API. Thus, the Newton runtime library needs to be linked against the host language program. Each Newton runtime library is specific to the Newton description provided so that the runtime library linked against the host language program has information about the Newton AST and the symbol tables for that Newton description. This means that for each Newton

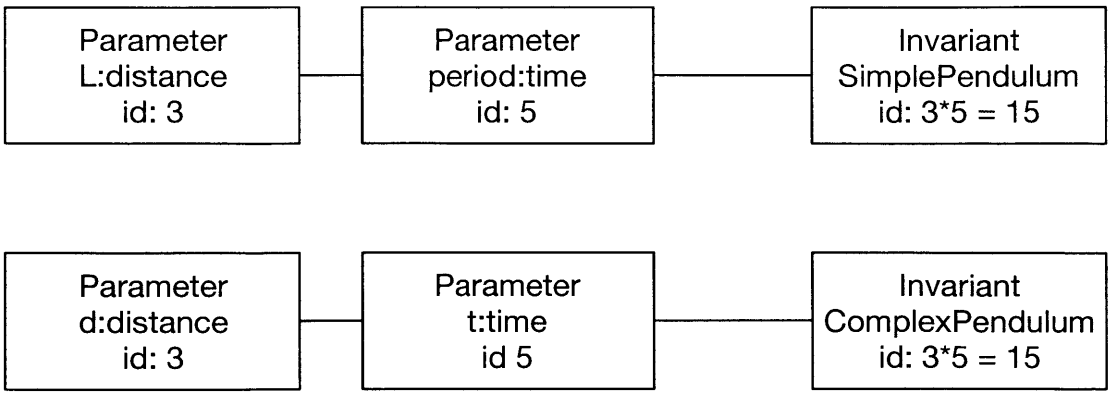


Figure 6-3: A Newton description cannot have two invariants with parameters containing the same set of Physics struct's.

description, there will be a unique Newton runtime library that holds information specific to that Newton description. At runtime, the host language program runtime can call the Newton library routine `newtonApiSatisfiesConstraints`.

As mentioned in the Introduction Chapter 1, in order for a piece of code to map to an invariant, it needs to meet the following four criteria.

1. The compiler of that program's language is able to pick an invariant out of all the invariants available that applies to a set of variables.

2. There is a mapping between the variables in the host language program and the parameters of an invariant.

3. Data types of the variables in the host language program reflect data types of invariant parameters.

4. The numerical values of the host language program variables should abide by the selected invariant.

The rest of this section describes how this transformation achieves each of the above four steps and changes the control flow of the host language program when invariants are violated at runtime.

6.2.1 Suggestions for Dynamic Tagging

In order to find the host language program variables whose values represent sensor signals, we can simulate the values coming from the sensors and observe how the values of the program variables are correlated with the simulated values. One framework that can be used to simulate the sensor variable values is Sunflower which takes in a file of values (called a trajectory file in Sunflower) that a particular sensor would return and a C program that reads from that sensor. Sunflower doesn't simply read from the trajectory file to return simulated sensor readings, but it interpolates what sensor data might be using the values written in the trajectory and a pre-defined sampling rate of the sensor. The Sunflower framework essentially allows us to simulate a sensor platform and a program reading from sensors on that platform. A command in the Sunflower framework called `valuestats` returns the value traces of the variables in a program binary meant to run on the target architectures Hitachi SH [25]. That is, if a host language program binary can run on Hitachi SH architecture, the Sunflower framework can be used to infer which host language program variables represent sensor signals.

Figure 6-4 demonstrates how trajectory files containing each sensor's hypothetical values can be used to select variables in a host language program that hold those values through the runtime of the program. We can define a file that contains arbitrary values for each sensor on the hardware and observe how the values of a variable change when the reader function of that sensor reads in the simulated values, which are interpolated values of the trajectory file. The `valuestats` function of the Sunflower framework lists value traces of C program variables so that we can compare them to the simulated values of a particular sensor. If all values of a host language program variable match the simulated values of a sensor, then the variable represents that sensor. See Figure 6-5.

Consider the pendulum swing count example from Chapter 4 reproduced here below.

```
1 acceleration@0 prevXacceleration = readFromXAccelerometer();
2 time durationInSeconds = 1000 * 30; // thirty seconds
3 time startTime = readFromSystemClock();
4
5 int swingCount = 0;
6
7 while (readFromSystemClock() < startTime + durationInSeconds)
8 {
9   acceleration@0 xAcceleration = readFromXAccelerometer();
```

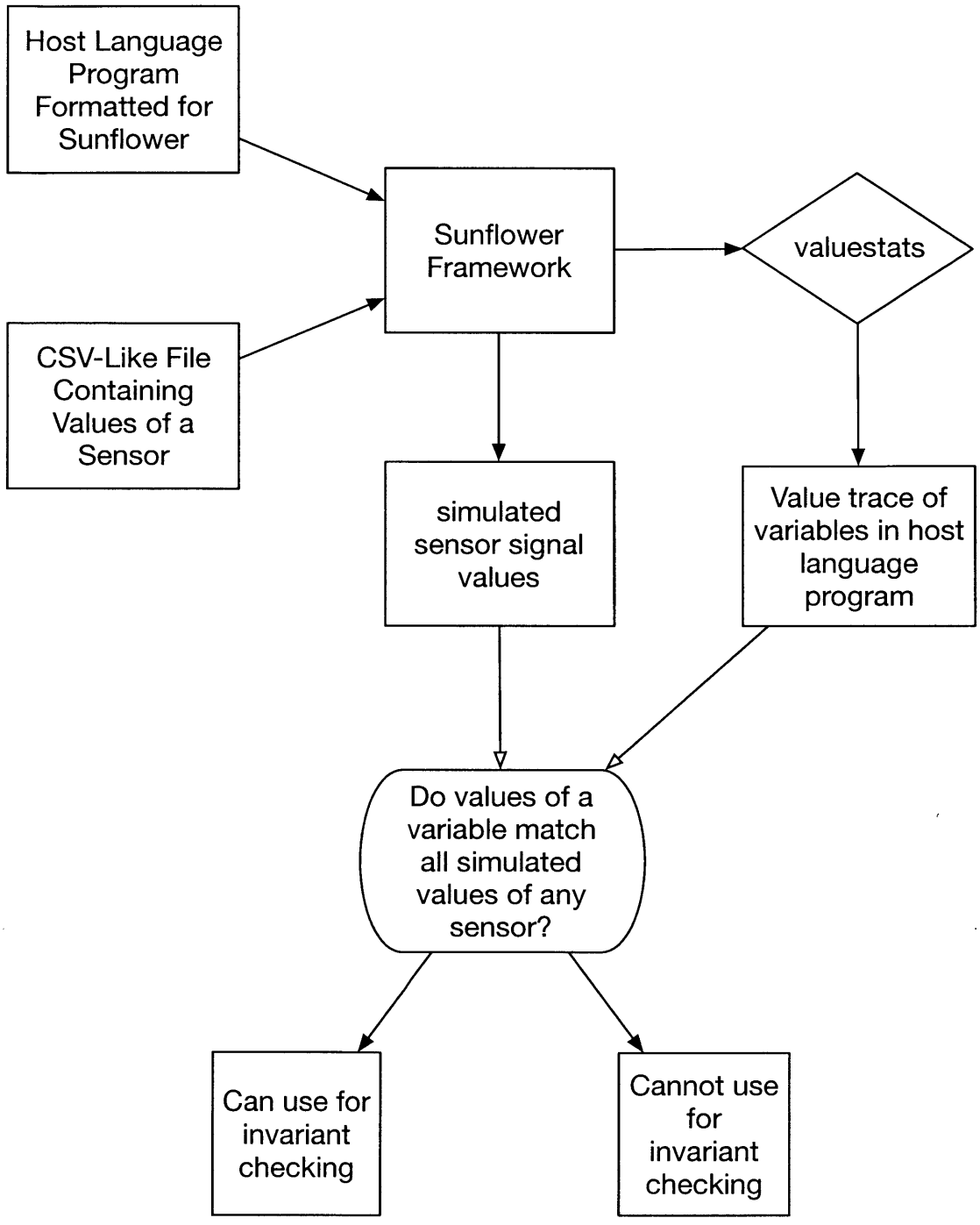


Figure 6-4: How the Sunflower Framework can be used to determine if a variable's values represent a sensor's values

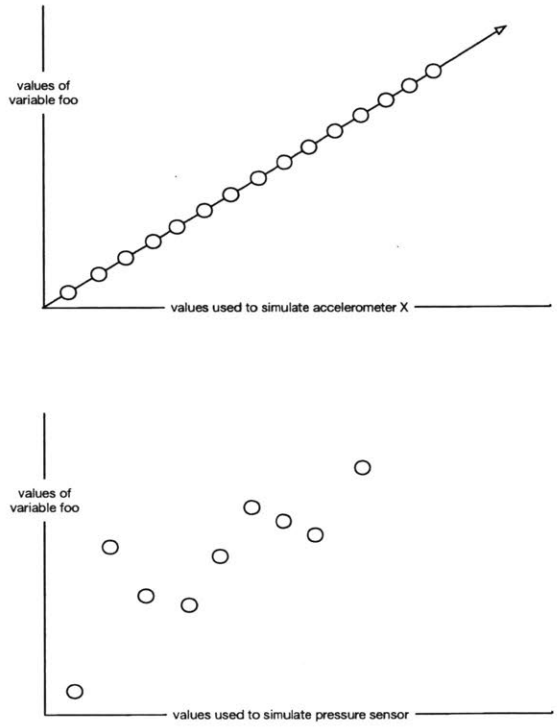


Figure 6-5: The graphs show the values of a variable in the output of the Sunflower framework's `valuestats` function. The first graph indicates a match between variable foo and accelerometer X.

```
10
11    if (prevXacceleration * xAcceleration < 0) {
12      swingCount++;
13    }
14
15    if (xAcceleration != 0) { // do not double count the change
16      prevXacceleration = xAcceleration;
17    }
18  }
19
20  printf("detected %d swings in the pendulum\n", swingCount);
```

The invariant SimplePendulum here takes in a variable of type acceleration, and in this code there are multiple variables of type `acceleration`, `xAcceleration` and `prevXacceleration`.

```
1  SimplePendulum : invariant(a: acceleration@0) =
2  {
3      a >= 2.2 * m / s ** 2,
4      a <=  10 * m / s ** 2,
5  }
```

The `acceleration` type variables `prevXacceleration` and `xAcceleration` in this code directly receive their values from the reader function `readFromXAccelerometer`, and therefore would hold values directly correlated with signal values coming from hardware sensor platform as indicated by dynamic tagging. As stated in the beginning of the thesis, the Newton API is interested in performing checks on the variables that represent sensor signals.

See the Sunflower framework manual for further information. This thesis has not explored whether it is possible to automate the process of formatting a host language program into a C program suitable as an input to the Sunflower framework. Dynamic tagging has not been implemented or tested in this thesis.

6.2.2 Suggestions for Static Tagging: Alternative to Dynamic Tagging

A host language can provide some syntax that would allow the programmer to annotate functions that would return values meant to be passed into a Newton invariant. All variables whose values are set to the return values of the annotated functions would then be marked as a potential parameter to the Newton runtime library call `newtonApiSatisfiesConstraints`. For example, the code below is annotated with "@sensor", and the host language compiler

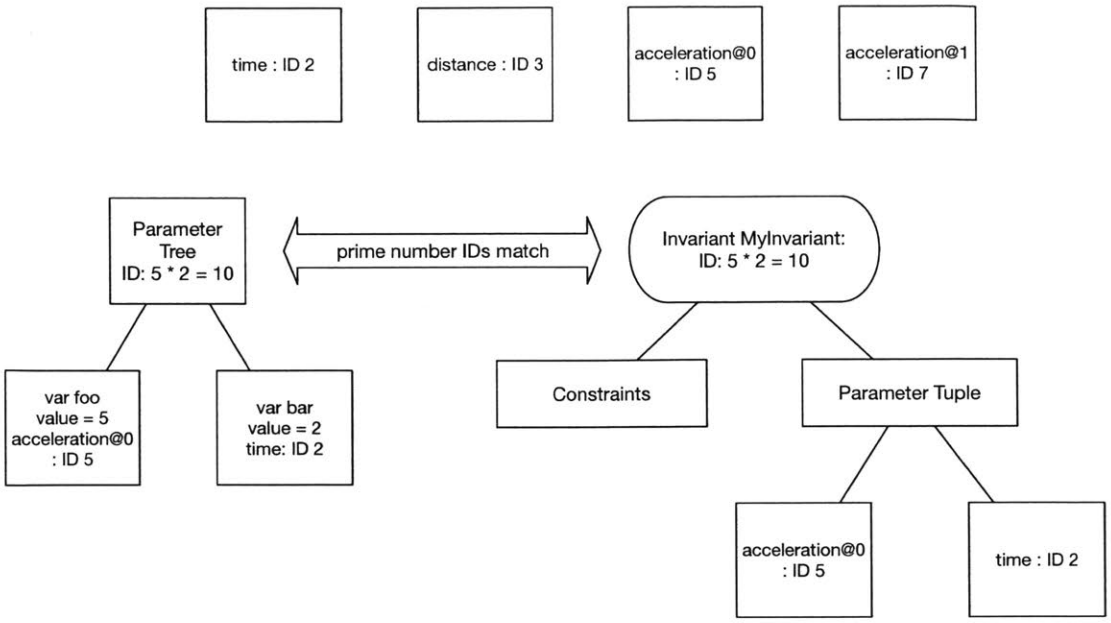


Figure 6-6: Every Physics type has a unique prime number ID associated with it. The invariant ID's and the parameter tree ID's are calculated by multiplying those ID's together. These ID's are used to find an invariant that should apply to a set of Physics type variables.

would recognize `read_from_accelerometer` as a special function that returns sensor signal values.

```
@sensor
acceleration read_from_accelerometer() {
    ...
}
```

All variables whose values are set by this function would then be a potential candidate for Newton invariant checking.

This static tagging approach is meant to substitute dynamic tagging, but it comes at a higher syntactic burden than using dynamic tagging. The programmer needs to worry about additional syntax added to the original host language syntax. The advantage of this approach is that it is simpler and less costly in terms of performance than dynamic tagging.

6.2.3 Constructing a Parameter Tree and Finding a Matching Invariant

This section explains how to find the correct invariant Newton subtree to walk for value checking, given a parameter tree comprising Physics variables. Section 5.3.2 explains how the Newton API method `newtonApiSatisfiesConstraints` actually performs that walk of the Newton tree.

To construct a mapping between a Newton invariant and a set of host language program variables, the Newton parser assigns a unique prime number ID for each of the base signals like `time` and `acceleration`, and it uses those prime number ID's to assign prime number ID's to the invariants in the Newton description. When the host language compiler builds its Newton AST of the source code, it calls `newtonApiGetPhysicsTypeByNameAndSubindex`, which returns a Physics struct containing a prime number ID associated with that Physics type. Therefore, the host language compiler can scan its Newton AST to aggregate a set of Physics variables whose multiplicative product of ID's match some invariant ID in the Newton description. This identification scheme allows quick look up of invariants that match a set of parameters. As such, having two different invariants which have parameters that have the same types is not allowed because they would map to the same ID number. For example, the host language compiler can find out through the Newton API that SimplePendulum invariant contains two parameters of types acceleration and time. See Figure 6-6 for an example.

In this thesis, the host language compiler only looks at the variables that can be accessed by a block (in this thesis, a function scope or a loop scope) to aggregate as parameters to Newton invariants. For example, if there are variables declared outside a particular block but used in that block, then those variables as well as the variables whose lifetime is inside the block are considered. Suppose that in a block, there are three variables of type `acceleration` and two variables of type `time`. Then, the host language compiler will aggregate 6 different sets of parameters to pass into `newtonApiSatisfiesConstraints`.

6.2.4 Mapping Host Language Variables to Invariant Parameters

In addition to mapping a set of parameters to an invariant, the Newton compile-time library needs to map each parameter within the set to its corresponding parameter in the invariant. An obvious way to achieve this mapping is to know what identifier is used in the Newton description. For example, in the invariant shown below, the parameter of type $acceleration$ has an identifier a. However, this approach requires the programmer to look up the identifier a in the Newton description.

```
1 PendulumInvariant : invariant(a: acceleration@0, period: time) =
2 {
3     a >= 2.2 * m / s ** 2,
4     a <=  10 * m / s ** 2,
5     period ~ (4 * Pi * Pi * pendulumLength / g) ** 0.5,
6     period >= 3 * second,
7     period <= 9 * second
8 }
```

Compilers of popular programming languages resolve this problem with calling convention, where each parameter is stored in the stack in reverse order by the caller, and the callee pops them off from the stack. This approach ensures that the parameters are in order.

The Newton library maintains the order of the parameters by simply numbering the parameters from the caller's side from 0 to $n - 1$ where n is the number of parameters, where the caller is a host language compiler and the callee is the Newton API. Similarly, the invariants defined in a Newton description file have their parameters numbered from 0 to $n - 1$. The mapping of the caller's parameter to the callee's parameter is then just matching the parameter numbers. Although one can imagine implementing something like a calling convention using a stack, the Newton API just involves the front end portion of the compiler, so storing this information in the Intermediate Representation was much simpler.

6.2.5 Transforming the Host Language Compiler's IR

We introduce an algorithm that describes the series of calls that a host language compiler can make to Newton in order to make that transformation in a given block. The two preconditions to running this algorithm are that

1. The source code is dimensionally consistent and that leaf identifier nodes in the

Newton AST with Physics types are marked as "NewtonTyped" as described in the Section 6.1.

2. Every variable in the source code IR are assigned exactly once and is defined before used.

Meeting the first precondition ensures that dimension types of a host language program are consistent. The second precondition makes use-def chain analysis to be done by a host language compiler simple in the transformation algorithm. Requiring every variable to be defined before used also serves another purpose. Consider the following code.

```
1 acceleration bar = read_from_accelerometer() * 5;
```

Suppose that a Newton invariant for this sensor platform takes in an acceleration as a parameter. Using dynamic tagging or static tagging, `bar` would not be a potential parameter to the Newton invariant even though the value it uses from `read_from_accelerometer()` should be checked by the Newton runtime library. If every variable is required to be defined before used, the above code would turn into the following.

```
1 acceleration foo = read_from_accelerometer();
2 acceleration bar = foo * 5;
```

Then, the transformation algorithm would select `foo` as a parameter for invariant checking. Notice that `foo` is a base variable of `bar`. Even though `bar` is not checked by the Newton runtime library, `foo` would be, and if `foo` failed the invariant checking, then it means any computation using `bar` would also be invalid.

Here is the algorithm. Suppose that a `Variable` is a data structure that represents a

variable in the host language IR.

Algorithm 10: Transforming Source Code AST Using Newton: Part 1 of 2

input : Newton AST A, a block subtree B in host language AST, array S of sensor types

output : Modified host language AST

Add to beginning of B an IR which encodes "valid = true;";

interestedVariables = { };

newIR = empty tree;

For all marked identifiers of A used or declared in B, add the corresponding identifier Variables in B to interestedVariables;

interestedVariables = runDynamicTagging(interestedVariables, S, B);

Continued.

Algorithm 11: Transforming Source Code AST Using Newton: Part 2 of 2

input : Newton AST A, a block subtree B in host language AST, array S of sensor types

output : Modified host language AST

sets of (sequences, matching newton invariant) =

findAllSequencesThatMatchNewtonInvariants(interestedVariables);

Add to newIR a Boolean Variable valid = true;

for *each sequence of variables and matching newton invariant* **do**

    newIR = empty tree;

    Add to newIR that says "valid = valid && newtonApiSatisfiesConstraints(newton, sequence)->satisfiesValueConstraint";

    Add to newIR that says "if not valid, execute the global error handler";

    Scan B for the point before the first usage of any of the variables in sequence but also where all assignment statements to variables in sequence have been completed. After this point, all accesses to variables in interestedVariables are reads. If this point is not found, then don't add this IR.

    Add newIR to the place found by scanning B;

end

First, the precondition of dimensional consistency needs to be met. At this point, the

host language compiler has a Newton AST of its source code.

Second, the host language compiler aggregates all variables used or declared in the block that may be potential parameters to Newton invariants.

Third, the host language compiler runs dynamic tagging step (or static tagging) as described in the previous section to all the variables in the current block. Despite the word "dynamic", the dynamic tagging step occurs at compile time. It is merely a simulation to help narrow down the set of variables that are candidates to be passed into the Newton API. The dynamic tagging step selects variables whose values directly represent the sensors on the sensor platform.

Fourth, the host language compiler finds every subsequence of the variables that match invariant parameter signatures (subsequence because the order of parameters matters). The cost of this step is analyzed in the next section. The order of the third and the fourth step, dynamic tagging and finding variable subsequences for invariants, can be interchanged since both steps together help narrow down the size of `interestedVariables`.

Fifth, find the point in the block where all of the variables in each subsequence are assigned a value. After that point, scan for the spot where any of the variables are first used, and then insert the Newton runtime library call `newtonApiSatisfiesConstraints` with all the variables in the subsequence passed in. The return value of this call is stored in another variable, which is also inserted during this transformation into the host language compiler IR.

Sixth, the host language compiler takes those results and modifies its own AST, to execute the global error handler before any of the variables that were passed in as an invariant parameter could be used, if the results indicate that the invariant has been violated. Whether the global error handler allows the program to return to the call site and continue executing code is up to the programmer. The end of Chapter 4 shows a sample global error handler that logs errors to a file. If the sensor platform has human operators, the global error handler could also trigger alerts to the user interface.

Providing syntax for the global error handler is not part of the Newton implementation. The host language must provide a way for the programmer to specify a handler function, through annotation or reserving a function identifier as a keyword. A syntax for annotation

may look like

```
@@error
void
globalErrorHandler(NewtonAPIReport* report) {
    ...
}
```

where the handler would take in the result of the Newton runtime library call `newtonApiSatisfiesConstraints` by default so that it can be used for better error messages.

Reserving a keyword for the global error handler function is same as how the identifier "main" in C is treated specially by typical C language runtimes.

To see the result of this transformation applied to the pendulum swing count code, see the end of Chapter 4.

6.2.6 Computation Cost

The computation cost of transformation with Newton consists of putting Newton AST in memory in the beginning of runtime (or serializing and deserializing Newton AST to call the Newton runtime library), performing dynamic tagging step, and finding all permutations of variables with Newton types that can match a Newton invariant signature. The dynamic tagging step as mentioned in the Section 6.2.1 uses the Sunflower framework which needs to perform I/O operations to trajectory files that list simulated sensor data. In addition, finding all sequences of variables with Newton types that can match any Newton invariant signature has a computation time that can grow exponentially with number of parameters per invariant and the size of the `interestedVariables` set. This step in theory takes exponential time, but the number of variables considered and the number of parameters in Newton invariants (< 5) tend to be small. For example, suppose that on average, there are N variables in a block that represent sensors, M Newton invariants, and L parameters per Newton invariant. For each of the M Newton invariants, we need to try N variables for each parameter spot to see if the type of the variable matches the type of the parameter. Finding the subsequences naively this way, in the worst case, will take $O(M(N/L)^L)$ time to find all subsequences. However, if N, M, and L are small, this step should take constant time in practice. If a short

compiling time is important, then transformation may not be ideal for programs with large number of Newton Physics type variables and complicated Newton invariants. In addition, this means the transformation must call the Newton API for all permutations of variables available, which slows down runtime performance as well.

6.2.7 Limitations

There are two limitations to the transformation that checks invariants at runtime on a list of variables. The limitations are primarily due to the way variables are selected to pass into the Newton API `newtonApiSatisfiesConstraints`, and this is the reason Newton invariants are meant to bind to variables that directly represent sensor values.

The first case occurs when dynamic tagging is unable to select a variable that is correlated with a single sensor signal because the variable is a combination of two or more sensor signals or because the variable holds a sensor value modified in some way (e.g. `read_from_accelerometer()` + 5). For example, if a function block only had "velocity foo = `read_from_accelerometer()` * `read_from_system_clock()`", the values of `foo` won't correspond to either `acceleration` or `time` signals. According to dynamic tagging as described in Section 6.2.1, the variable won't be selected as a candidate to pass into a Newton invariant because the values are not exact matches of simulated sensor values on the Sunflower framework. Even if an invariant signature has a velocity as a parameter, the dynamic tagging would not select it if a sensor on the hardware does not directly measure velocity. There are too many possibilities of how two or more sensor signals can be combined to make a new variable that needs to be checked using Newton. Suppose that a Newton invariant takes in an acceleration sensor value, but the variable holds a value 2 times the acceleration. As discussed in the previous section, dynamic tagging helps reduce the set of variables to only those that are directly correlated with each of the sensors on the hardware platform, but this means the transformation may have missed the set of variables that are correlated with two or more sensor readings at the same time.

The second case occurs when a Newton invariant is unable to take in a parameter that is a derivative of the Physics types of the sensor variables. In other words, if a sensor

platform has only accelerometers, the Newton invariant signature cannot have a velocity as a parameter. Newton invariants, therefore, have to take parameters that are assumed to hold direct sensor reading values. However, Newton invariants can be written in a way that could put constraints on variables of Physics types that are time derivatives of sensor variables if the system clock is available and taken in as one of the parameters of the Newton invariants. For example, to impose constraints on velocity, a Newton invariant could take in `a : acceleration` and `t : time`, which represent acceleration sensor and system clock sensor respectively and then put constraints on `a * t`, like this $5 * m/s <$ `a * t`.

6.3 Transformation to Reduce Sensor Redundancy

The previous transformation resulted in a host language program that made a call to `newtonApiSatisfiesConstraints` to check invariants at runtime. In contrast, this transformation takes advantage of a specific type of invariants, ones that show sensor redundancy, to remove redundant code in the host language program at compile-time. Removing repetitive code that works with redundant sensors will eliminate the need to perform read/write operations with duplicate sensors and thus decrease the number of registers needed at runtime as well as decrease the number of memory operations. The benefit of this transformation is reducing the code size and freeing up registers that were previously storing values from redundant sensors. The disadvantage is precisely the cost of not having sensor redundancy – lost reliability. For mission critical systems with low tolerance for errors, removing sensor redundancy may not be desirable, but for systems where performance is more important, this transformation can be helpful.

The preconditions for this algorithm are the same as those for the previous transformation.

1. The source code is dimensionally consistent and that leaf identifier nodes in the Newton AST with Physics types are marked as "NewtonTyped" as described in the Section 6.1.

2. Every variable in the source code IR are assigned exactly once and is defined before

used.

The algorithm for exploiting sensor redundancy is as follows:

Algorithm 12: Transforming Source Code AST Using Newton to Eliminate Sensor Redundancy. Part 2 of 2

input : host language AST A, Newton AST of the host language program N, array T of sensor types

output : Modified host language AST

1. Ask the Newton compile-time library method `newtonApiGetRedundantSensors` if there are any sets of redundant sensor types specified in the Newton description. Specifically, let a set of redundant sensors be S_i. The Newton API returns U_S, a set of all S_i's. Each $s_j \in S_i$ returned from the call is a Physics type.
2. Find out which variables in the entire host language program match Physics types s_j in S_i. Let this variable set be V_i. Each $s_j \in S_i$ is the Physics type token used in a type expression of the assignment statement of variable $v_k \in V_i$. We use subscript k because the size of V_i and the size of S_i may be different.
3. Run dynamic tagging with Sunflower framework (or static tagging) to find out if the variables in the program that matched, $v_k \in V_i$, actually represent readings of sensors $t \in T$. Match simulated sensor values to simulated variable values only if the values match exactly. In other words, if none of the sensors in the array $t \in T$ can satisfy $v_k.value = t.value$ for all simulated values of t on the Sunflower framework, eliminate v_k from the set V_i.

Continued.

Algorithm 13: Transforming Source Code AST Using Newton to Eliminate Sensor Redundancy. Part 2 of 2

input : host language AST A, Newton AST of host language program N, array T of sensor types

output : Modified host language AST

3. Let the set of RHS expressions of assignment statements of variables in V_i be R_i, where RHS of v_k assignment statement is r_k. The size of R_i and the size of V_i are same because of the second precondition. Substitute r_0 into r_1 through r_n in the host language AST A, where n is the size of R_i. This step is correct because all $\forall v_k \in V_i, v_k.value = r_k.value$ and $v_k.value = t_k.value$ so that all r_k in R_i have same values.
4. Similarly, substitute s_0 into s_1 through s_n in N. After this step, $s_0 = s_1 = ... = s_n$ and $r_0 = r_1 = ... = r_n$. This step won't affect type expressions in A because A just uses floats for all Physics types.
5. Since the same expression is used for all r_k in R_i, this will aid Common Subexpression Elimination in the C compiler optimization phase.
6. Since all v_k in V_i are mapped to the same RHS, this will aid Global Value Numbering in C compiler optimization phase.
7. Repeat steps above for all S_i in U_S.

The Newton compile-time library method called in the first step of this algorithm `newtonApiGetRedundantSensors` is not currently implemented, but we present a possible implementation of the method here.

Algorithm 14: Newton API: identifying Physics types that represent redundant sensors

input : a Newton AST built from parsing a Newton description

output : Set of all sets containing redundant sensor types

1. Initialize every invariant parameter type as its own set of equivalent sensor types.
2. For every constraint that says `A` $\sim$ `B`, where `A` and `B` are invariant parameters, union the set to which type of `A` belongs with the set to which type of `B` belongs.
3. Return all sets that has more than one element.

The equivalent parameter case in Step 2 can easily be implemented by checking recur-

sively that LHS and RHS expressions each have only only one invariant parameter as their only factor.

Now, we show an example of this transformation along with the Newton description used. Imagine that extra reliability gained through redundant sensors was not needed. The following C code with Newton Physics type syntax is a simple sensor voting logic inside a sensor reader function where each sensor "votes" by contributing to an average calculation.

```
1     pressure@1 first_pressure = read_from_pressure_sensor1();
2     pressure@2 second_pressure = read_from_pressure_sensor2();
3     pressure@3 third_pressure = read_from_pressure_sensor3();
4
5     pressure@0 average_pressure = (first_pressure + second_pressure +
      third_pressure) / 3;
6     return average_pressure;
```

Suppose the Newton description has an invariant that states the following.

```
1 PressureSensorRedundancy: invariant(
2     pressure@1 pressure_front_sensor,
3     pressure@2 pressure_back_sensor,
4     pressure@3 pressure_side_sensor
5 ) = {
6     pressure_front_sensor ~ pressure_back_sensor,
7     pressure_back_sensor ~ pressure_side_sensor
8 }
```

The Newton description tells us that three pressure sensors of types `pressure@1`, `pressure@2`, and `pressure@3` are redundant. The Newton API method `newtonApiGetRedundantSensors` would initialize three sets of Physics types, each containing `pressure@1`, `pressure@2`, and `pressure@3`. Upon encountering the first constraint, `pressure_front_sensor` $\sim$ `pressure_back_sensor`, the two sets containing `pressure@1` and `pressure@2` are joined, and after the second constraint, `pressure_back_sensor` $\sim$ `pressure_side_sensor`, `newtonApiGetRedundantSensors` returns one set containing all three pressure types `pressure@1`, `pressure@2`, and `pressure@3`.

Now we can follow the redundancy elimination transformation described in this section. Let V_i as described in the algorithm consist of `first_pressure`, `second_pressure`, and `third_pressure`, and let R_i consist of `read_from_pressure_sensor1()`, `read_from_pressure_sensor2()`, and `read_from_pressure_sensor3()`. After Step 3 of the algorithm, which unifies all RHS of equivalent sensor variables' assignment state-

ments, the C compiler's IR now encodes the following. Remember that C compiler's IR encodes Physics types as floats.

```
1    float first_pressure = read_from_pressure_sensor1();
2    float second_pressure = read_from_pressure_sensor1();
3    float third_pressure = read_from_pressure_sensor1();
4
5    float average_pressure = (first_pressure + second_pressure +
     third_pressure) / 3;
6    return average_pressure;
```

After Step 4, which retains the Physics types, the Newton IR of the original program encodes the following.

```
1    pressure@1 first_pressure = read_from_pressure_sensor1();
2    pressure@1 second_pressure = read_from_pressure_sensor1();
3    pressure@1 third_pressure = read_from_pressure_sensor1();
4
5    pressure@0 average_pressure = (first_pressure + second_pressure +
     third_pressure) / 3;
6    return average_pressure;
```

If this redundancy elimination transformation is performed before the C compiler's own optimizations, then Common Subexpression Elimination of C compiler's IR will result in the following.

```
1    float temp = read_from_pressure_sensor1();
2    float first_pressure = temp;
3    float second_pressure = temp;
4    float third_pressure = temp;
5
6    float average_pressure = (first_pressure + second_pressure +
     third_pressure) / 3;
7    return average_pressure;
```

If the C compiler performs Value Numbering and Dead Code Elimination on this code, then it is simplified even further.

```
1    float temp = read_from_pressure_sensor1();
2
3    float average_pressure = (temp + temp + temp) / 3;
4    return average_pressure;
```

This pressure sensor reader method now contains no redundancy. This compile-time transformation eliminates redundancy for a sensor platform that has redundant sensors. An example use case of this transformation might be a smart watch with redundant sensors that use a lot of battery power. If having accurate data from redundant sensors, say for example accelerometers, on the platform is not safety-critical, then this transformation may help improve performance of the system.

Chapter 7

Evaluation

As mentioned in the thesis proposal, this project is evaluated based on whether the following tasks can successfully be completed.

1. Given a Newton description, an API call on a physics quantity should return a correct SI unit.

2. Given a Newton description, an API call on a physics expression should return a correct SI unit.

3. Newton is expressive enough to describe realistic systems. This MEng project will compile at least 12 different Newton description files describing real-world systems and verify the correctness of each description by checking the SI units of all Physics types in the description.

4. It is possible to develop examples of pseudocode for 12 or more algorithms which would benefit from having an implementation in a host language whose compiler uses the Newton API.

Furthermore, this thesis accomplishes additional goals since the thesis proposal, namely describing transformation algorithms using the Newton API and their use cases in example systems.

The performance analysis of the two Newton API methods `newtonApiInit` and `newtonApiSatisfiesConstraints` are shown below. The first table shows the mean

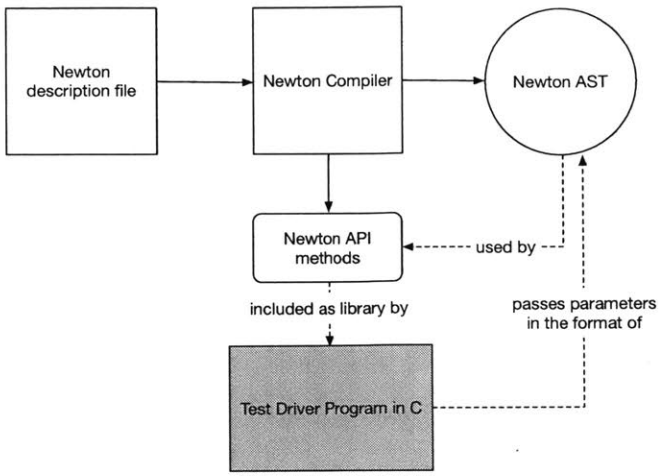


Figure 7-1: To test the Newton API, a test driver program will construct a Newton AST and call the Newton API on that tree. The test driver programs will have access to the Newton API by including its header file as library.

runtimes (in nanoseconds) of both methods for each of the 12 Newton descriptions over 40 iterations, and the second table shows the standard deviations of those runtimes.

Newton Description	newtonApiInit	newtonApiSatisfiesConstraints
Step Counter	1503298	139392
Activity Classifier	1339013	156423
Vehicle Distance	675434	38847
Weather Balloon	1032082	89110
Airplane Pressure	684599	51114
Ball Dropped	1027406	77935
GPS Walking	786148	60469
Jet Engine	1010725	109836
Motor Wheel Chair	972281	63757
Reactor Rod	68395	62814
Sensor Life	1006878	88640
Tire Pressure	1621634	142686

Newton Description	newtonApiInit	newtonApiSatisfiesConstraints
Step Counter	277499	19956
Activity Classifier	382776	44034
Vehicle Distance	249525	29003
Weather Balloon	340519	44869
Airplane Pressure	309749	35106
Ball Dropped	351014	37800
GPS Walking	291808	40901
Jet Engine	355701	46060
Motor Wheel Chair	290080	34842
Reactor Rod	55412	36288
Sensor Life	479864	45578
Tire Pressure	455739	26999

Chapter 8

Future Work and Challenges

This thesis has focused on the Newton API, its implementations, and designing how a compiler may interact with the API, as described in the thesis proposal. However, we do not yet have a compiler that uses the Newton API to implement the algorithms mentioned in Chapter 6. A main task that is to be completed in the future is implementing a compiler that actually uses the Newton API and analyzing the effectiveness of the compiler's output.

The Newton API as of now only supports very simple host language programs that comprise binary expressions and assign statements. Making more language constructs will help support a wider variety of host language programs. Another feature that would help solidify the Newton syntax is allowing definitions of vectors and their operations. For example, Newton currently views variables defined in the Newton descriptions as a Physics variable containing a scalar value. As of now Newton supports having multiple dimensions for a Physics variable, such as 3 axes for acceleration. Having vectors would allow encapsulation of these 3 variables into one vector as well as operations on these vectors such as magnitude, dot products, and cross products.

In the current syntax of Newton, multiple invariants cannot have the same parameter tuple signatures. As of now, when the Newton API looks for an invariant that matches a list of parameters passed in from the host language compiler, it looks at the first invariant that matches and checks if that invariant satisfies. A feature that may enforce the specification that invariant signatures must be unique would be a mitigation pass that checks that all invariant signatures are unique at the Newton description parsing step.

Newton's type system can be expanded to include operations like type casting so that programmers can do computations on variables with different Physics types.

As mentioned in 5, if a programmer wants to execute some code path after the Newton API call returns an error, the host language compiler needs to support additional syntax. This will not require any interaction with the Newton API, but it is up to the compiler writer to include this flexibility for the programmer.

Another feature that would increase the expressiveness of Newton is logical operators on the constraints. There are examples where the sensor values are related to each other by something other than laws of physics, and logical operators allow expressions such as "If sensor A has this value, sensor B must have that value." For example, when an airplane is landing, its wings must be perfectly level. In other words, if the altitude sensor indicates that it is close to the ground, but the gyros of the plane indicate that the body of the plane is tilted, then the sensor values are abnormal, and the pilot should be notified right away by the airplane software. The airplane's speed is constrained by its altitude as well. If the airplane is low to the ground, there are ranges of velocity acceptable for a particular altitude [2]. There are examples in transmission gears of cars where the transmission range sensor in the Power Train Control Module indicates what gear the car is in [5]. Depending on the gear position of the car, the maximum engine speed may differ, where the engine speed is read from another sensor. Note that in these cases, the sensor values are still constrained by some relationships, but not by laws of physics.

The transformation to check invariants at runtime assumes a global error handler when invariants are violated at runtime (see Section 6.2). If there is a clean syntax to annotate which functions should handle which invariant failures, different functions could be used to handle each failure case, which could help make the system more robust.

Finally, the transformations themselves can be part of the Newton API if there are methods that take in LLVM IR's and transform them according to the algorithms mentioned in this thesis. Achieving this task will shift the responsibility of the host language compiler to Newton implementation and thus make Newton easier to use.

Chapter 9

Summary

Newton is a language for describing the laws of physics. A compiled Newton description file and the Newton libraries can be used by any host language compiler to be able to perform type checking and transform the the source code to be able to perform runtime value checking. Being able to run these checks is useful because the hardware manufacturer can enforce that certain relationships are preserved among sensor signal values on a hardware platform. Much of Newton's research contribution is being able to bind variables in a host language to specific constraints through the Newton API. For this thesis, we have implemented the Newton API, which is an interface to any compilers that may want to use it for the purposes of programming on platforms with sensors.

Appendix A

Appendix: Transformed Examples

In this section, we will explore the transformations that can be applied to the real-world examples mentioned in Chapter 5. Each section in this chapter illustrates a host language program that may utilize the Newton description and how it may be transformed. To see the Newton description for each example, refer to Section 5.5.

All twelve examples in this appendix include the transformation to check invariants. Pedometer Step Counter, Activity Classifier, GPS Walking, and Sensor Life examples include the transformation to reduce sensor redundancy.

A.1 Pedometer Step Counter

The following code is a C program with additional syntax to utilize Newton that implements the step counter algorithm described in [7]. The main method of the code is the `run_step_counter()` method which starts reading from the accelerometer of the step counter. The algorithm uses a linear shift register and a dynamic threshold to determine where a step happens. The linear shift register contains two registers `sample_new` and `sample_old`. For every new acceleration data `sample_result`, the `sample_new` is shifted to `sample_old`, and `sample_result` is stored in `sample_new` if the difference between `sample_result` and the previous `sample_new` value is greater than some precision threshold. A step is detected when there is a negative slope in the acceleration, which happens when `sample_new` is less than `sample_old` and when the graph crosses below the dynamic

threshold. The dynamic threshold is the half the value of the minimum and the maximum of the last 50 sample acceleration data. In the code, an array of last 50 acceleration data is used to calculate the dynamic threshold, and `most_recent_index` is the location in the 50 samples array that contains `sample_new`.

```
/*
 * http://www.analog.com/media/en/technical-documentation/technical-articles/pedometer.pdf
 *
 * This code is a hypothetical implementation of pedometer step counter as described in the paper above.
 *
 */
#include <stdio.h>
#include <stdint.h>

#define SAMPLE_SIZE 50
#define PRECISION_THRESHOLD 0.1

enum RegulationMode { SEARCHING, FOUNDOUT };

void run_step_counter()
{
    /* initializations */
    int step_count = 0;
    int most_recent_index = 0;
    int sample_size = 0;
    time last_step_recorded = 0;
    bool array_is_full = false;
    acceleration precision_threshold = PRECISION_THRESHOLD;
    acceleration dynamic_threshold = 0;
    acceleration samples[SAMPLE_SIZE]; /* x[most_recent_index] is sample_new from the paper*/
    acceleration total = 0, threshold = 0;
    RegulationMode mode = SEARCHING;

    /* start reading from step counter */
    while (true)
    {
        acceleration sample_old = samples[most_recent_index]; /* sample_old */

        time sample_time = get_new_sample_results_and_filter(x, y, z, most_recent_index, precision_threshold);
        most_recent_index = most_recent_index + 1 % SAMPLE_SIZE;

        array_is_full = most_recent_index == 0;
        sample_size = array_is_full ? SAMPLE_SIZE : most_recent_index;

        dynamic_threshold = get_dynamic_threshold(samples, sample_size);

        if (samples[most_recent_index] < sample_old && sample_old > dynamic_threshold && samples[most_recent_index] < dynamic_threshold)
        {
```

```
44              time step_duration = sample_time - last_step_recorded;
45
46              /* time window */
47              if (step_count == 0 || (step_duration > 0.2  && step_duration <
    2))
48              {
49                  step_count++;
50                  last_step_recorded = sample_time;
51
52                  if (step_count == 4)
53                  {
54                      mode = FOUNDOUT;
55                  }
56              }
57              /* perform count regulation. invalid step discovered */
58              else
59              {
60                  mode = SEARCHING;
61                  step_count = 0;
62              }
63          }
64      sleep(500);
65      }
66  }
67
68  time get_new_sample_results_and_filter(
69      acceleration samples[SAMPLE_SIZE],
70      int most_recent_index,
71      acceleration precision_threshold
72  ) {
73      time now = read_system_clock(); /* timestamp for now */
74
75      /*
76       * sample_result from the paper. Simulate adding all the inputs through a
     summing unit
77       */
78      acceleration@0 x = read_from_x_accelerometer();
79      acceleration@1 y = read_from_y_accelerometer();
80      acceleration@2 z = read_from_z_accelerometer();
81      acceleration new_sample = x + (acceleration@0) y + (acceleration@0) z;
82
83      /*
84       * Each sensor values will be evaluated for Newton invariant preservation
85       * at every read. set the global variable VALID to false if invariant not
     satisfied.
86       */
87
88      if (abs(samples[most_recent_index] - new_sample) > threshold)
89          samples[most_recent_index] = new_sample;
90
91      return now;
92  }
93
94  /*
95   * Get the middle value of the last 50 samples
96   */
```

```
acceleration get_dynamic_threshold(
    acceleration samples[SAMPLE_SIZE],
    int sample_size
) {
    acceleration min = LONG_MAX, max = LONG_MIN;

    for (int index = 0; index < sample_size; index ++)
    {
        if (samples[index] < min)
            min = x[index];

        if (samples[index] > max)
            max = samples[index];
    }

    return (min + max) / 2;
}

void update_average_acceleration(
    acceleration@0 samples[SAMPLE_SIZE],
    acceleration@0 *average,
    int sample_size
) {
    *average = 0;
    for (int index = 0; index < sample_size; index++)
    {
        *average += samples[index];
    }
    *average /= SAMPLE_SIZE;
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    ConstraintReport* currentConstraint = report->firstConstraintReport;
    while (currentConstraint != NULL) {
        /* log errors in some file */
        currentConstraint = firstConstraint->next;
    }
}
```

The following code illustrates a series of the Newton compile-time library calls made by the C compiler to be able to make the invariant-checking transformation described in 6.2 for the method `run_step_counter`. It illustrates what is done by the host language compiler for dimensional type-checking and IR transformation. Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

```
    /* Called in the beginning of host language compiler parser phase */
    State * newton = newtonApiInit("pedometer.nt");

    /* When the C compiler sees the token "time" for a type declaration in "
    time last_step_recorded = 0;"*/
  IrNode * lastStepRecorded = makeIrNodeSetValue(
    newton,
```

```
    kNewtonIrNodeType_Tidentifier,
    "time",
    0
  );
  lastStepRecorded->physics = newtonApiGetPhysicsTypeByNameAndSubindex(newton
    , "time", 0);

    /* The C compiler can initialize values if known at compile-time. */
    lastStepRecorded->value = 0;

    /* Add this Newton AST node to inside a struct where the C compiler keeps
     track of variables, say Variable */
    Variable * varTime = /* find Variable in C compiler's IR that corresponds
     to lastStepRecorded */ ;
    varTime->newtonNode = lastStepRecorded;

    /*
     * Repeat for acceleration when the C compiler sees "acceleration
    dynamic_threshold = 0"
     * The C compiler can pass in a default value for sub-dimension. Here, X
    axis is the default.
     * If the C compiler reads "acceleration@1", pass in 1 for the sub-
    dimension parameter in newtonApiGetPhysicsTypeByNameAndSubindex
     * Do the same for every Physics type token seen and construct Newton IR.
     */

    /*
     * "time last_step_recorded = 0;"
     *
     * Whenever a statement or an expression contains a type defined by
    Newton, the C compiler will
     * construct a Newton AST of a statement or an expression at compile time
     to verify dimensional consistency.
     */
  IrNode *  leftTerm = genIrNode(newton,  kNewtonIrNodeType_PquantityTerm,
    lastStepRecorded /* left child */,
    NULL /* right child */,
    NULL /* source info */);
  IrNode *  leftExpression = genIrNode(newton,
    kNewtonIrNodeType_PquantityExpression,
    leftTerm /* left child */,
    NULL /* right child */,
    NULL /* source info */);
    IrNode * zero = makeIrNodeSetValue(newton,  kNewtonIrNodeType_Tnumber,
      NULL,
      0);
  IrNode *  rightTerm = genIrNode(newton, kNewtonIrNodeType_PquantityTerm,
    zero /* left child */,
    NULL /* right child */,
    NULL /* source info */);
  IrNode *  rightExpression = genIrNode(newton,
    kNewtonIrNodeType_PquantityExpression,
    rightTerm /* left child */,
    NULL /* right child */,
    NULL /* source info */);
  IrNode *  statement = genIrNode(newton,
```

```
    kNewtonIrNodeType_PquantityStatement,
53    leftExpression /* left child */,
54   rightExpression /* right child */,
55   NULL /* source info */);
56
57   ConstraintReport* dimensionReport = newtonApiDimensionCheckTree(newton,
    statement);
58   if (! dimensionReport->satisfiesDimensionConstraint)
59   {
60       /* If dimensional consistency is violated, then the C compiler should
     add this to compile errors*/
61   }
62
63   /*
64    * Repeat the steps above for the entire program pedometer.c
65    * By this step, we know if expressions and statements in pedometer.c are
     dimensionally consistent.
66    */
67
68   /*
69    * Add all variables declared or used in the current block to the current
     block.
70    */
71   Block* block = .... // or whatever struct C compiler uses for block
72   block.addVar(...);
73
74   /*
75    * Run dynamic tagging. (or static tagging)
76    * After tagging is over, each variable's isTagged flag is set.
77    */
78  tagVariablesCorrelatedToSignals(newton, block->variables);
79
80   /*
81    * Of all possible variables that are tagged, find out which subsequences
     of variables can actually
82    * be passed into an invariant (matches an invariant signature in order)
83    *
84    * If a variable cannot be passed into an invariant, turn off it's
    isTagged flag
85    * That means every variable in the returned variable set has isTagged
    flag set.
86    */
87   VariableListAndInvariantTuple* variableListAndInvariantHead =
    getAllPossibleSubsequencesOfParameters(block->taggedVariables, newton->
    invariantList);
88
89   /*
90    * Construct parameter trees for each subsequence of variables that can
    be passed in
91    */
92   while (variableListAndInvariantHead != NULL) {
93       Variable* var = variableListAndInvariantHead->variableHead;
94     IrNode *  root = genIrNode(newton,  kNewtonIrNodeType_PparameterTuple,
95       NULL /* left child */,
96       NULL /* right child */,
97       NULL /* source info */);
```

```
98
99          while (var != NULL) {
100             head->type = kNewtonIrNodeType_Pparameter;
101             newtonApiAddLeafWithChainingSeqNoLexer(newton, root, var);
102             var = var->next;
103         }
104
105         newtonApiNumberParametersZeroToN(newton, root);
106
107         var = variableSetHead->variableHead;
108         while (var != NULL) {
109             /* first parameter tree appended is to be passed into the first
    invariant appended, and so on */
110             appendParameterTreesToVariable(var, root);
111             appendInvariantToVariable(var, variableSetAndInvariantHead->
    invariant);
112             var = var->next;
113         }
114
115         var = variableListAndInvariantHead->next;
116     }
117
118     /*
119      * Insert code into C compiler's AST that would call
120      * newtonApiSatisfiesConstraints(newton, var->parameterTreeHead, var->
    invariant),
121      * where parameterTreeHead is the constructed parameter and var->
    invariant is the invariant that
122      * corresponds to the parameter tree.
123      * Note that passing in the invariant pointer is merely an optimization.
124      *
125      * This means that whenever a sensor value changes, the Variable
    corresponding to that sensor
126      * should update its value as well if it is tagged.
127      */
128     VariableListAndInvariantTuple * var = variableListAndInvariantHead;
129     while (var != NULL) {
130         IrNode* parameterTreeRoot = var->parameterTreeRoot;
131         Invariant* invariantRoot = var->invariantRoot;
132         while (parameterTreeRoot != NULL && invariantRoot != NULL) {
133             /*
134              * Find the place in the block where all of the variables in
    parameterTreeRoot have been defined but
135              * none of them have been used yet.
136              * add code that calls newtonApiSatisfiesConstraints(newton,
    parameterTreeHead, invariantRoot) to this place in host language IR;
137              */
138
139             parameterTreeRoot = parameterTreeRoot->next;
140             invariantRoot = invariantRoot->next;
141         }
142         var = var->next;
143     }
```

Now the body of the step counter code can be transformed into the following by the C

compiler if we execute the global handler when the invariant is violated at runtime. The transformation of the method `get_new_sample_results_and_filter` is shown below. To facilitate presentation, only `x`, `y`, and `z` are shown to be declared before used although all variables should be declared before used for this transformation.

Note here that this code assumes that type casting feature when it is adding variables of distance types. The current implementation does not allow adding variables of different axes because different sub-dimensions are treated as separate Physics types. This can be resolved by allowing type casting on individual Physics expressions.

```
time get_new_sample_results_and_filter(
    acceleration samples[SAMPLE_SIZE],
    int most_recent_index,
    acceleration precision_threshold
) {
    VALID = true;
    time now = read_system_clock(); /* timestamp for now */

    acceleration@0 x = read_from_x_accelerometer();
    acceleration@1 y = read_from_y_accelerometer();
    acceleration@2 z = read_from_z_accelerometer();

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree of containing x, y, and z*/);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    /*
     * sample_result from the paper. Simulate adding all the inputs through a
     summing unit
     */
    acceleration new_sample = x + (acceleration@0) y + (acceleration@0) z;

    /*
     * Each sensor values will be evaluated for Newton invariant preservation
     * at every read. set the global variable VALID to false if invariant not
     satisfied.
     */

    if (abs(samples[most_recent_index] - new_sample) > threshold)
        samples[most_recent_index] = new_sample;

    return now;
}
```

There are two Newton invariant mentioned in Section 5.5: one invariant states that the tangential velocity of the pedometer worn must be roughly equal to the radius of the smart-watch pedometer times the angular velocity, and another invariant states

the acceleration maximum and minimum values. In the transformation of the method `get_new_sample_results_and_filter` above, the variables `x`, `y`, `z` are valid parameters to the second invariant but not to the first. They also directly represent the sensor variables.

In addition to the invariant checking transformation above, we show the sensor redundancy transformation here. Imagine that the hardware manufacturer of the pedometer wants the pedometer to have a low-power option which would execute the code that does not deal with redundant sensors. As shown in Section 6.3, this transformation changes the sensor reader function, which is the interface between the software and redundant sensors. Suppose that these sensor reader functions just takes the average of the two redundant sets of acceleration and gyro sensors.

```
acceleration read_from_accelerometerX() {
    acceleration@0 average = (read_from_accelerometerX1() + (acceleration@0)
    read_from_accelerometerX2()) / 2;
    return average;
}
```

After applying the transformation that reduces sensor redundancy, the above code becomes the following.

```
acceleration read_from_accelerometerX() {
    acceleration@0 average = (read_from_accelerometerX1() +
    read_from_accelerometerX1()) / 2;
    return average;
}
```

The same transformation can be applied to the reader functions of other acceleration axes data and gyro data as well. Whether this performance gained is worth the reliability lost is not investigated in this thesis.

A.2 Activity Classifier

This section is about a piece of code describing the system in Section 5.5.2. This example is different because it is written in Python, which means there is no compile-time type checking. Python runtime should call `newtonApiDimensionCheckTree` at the earliest time the types of variables become available. Note that the Python compiler cannot observe a type word that would trigger a Newton API call since types of variables are unknown until runtime. To include the Physics in Python syntax at all, there needs to be a way for the

Python programmer to declare those types. In this case, we just use Python's syntax for instantiating class objects, but that is entirely up to the Python compiler writer.

This code has three main parts. The first part defines sensor reader functions that simply read from sensors of a smart-watch and return the values according to Python 3 function type annotation [13]. Function type annotation reduces the amount of dynamic typing done in Python but is not essential as Python can perform duck-typing. The code only defines the method for the x acceleration, but suppose other reader functions are defined similarly. The second part is a function that collects sensor data for a certain time duration. This data is then used in the third part, which returns a classifier from the sensor data.

Suppose that the person wearing the smart-watch is performing a specified daily activity under a laboratory condition so that a file containing the labels of the activities performed is readily available to be passed into the classifier. Data collection and making the classifier happens inside a smart-watch which contains various sensors, a processor, and a memory.

```
import csv
import numpy as np
from sklearn.neural_network import MLPClassifier

# Python 3 type annotation. See https://docs.python.org/3/library/typing.html
def read_from_acceleration_x() -> acceleration@0:
    acceleration@0 x = read_from_some_register_on_sensor_platform()
    return x

# assume all sensor reader methods are defined by hardware manufacturer like
    x accelerometer

# Here, a user wearing a smart watch with sensors is performing some pre-
    defined
# actions listed in a file called label_filename
def collect_data():
    feature_matrix = []
    while read_from_system_clock() < 10000000:

        # Here, there are reads from multiple sensor types in the same block.
     The generated code from
        # Python compiler will pass in these variables and Newton will
    perform checks on them based on
        # activity_classification_pedometer.nt
        #
        # the values read for each sensor. Each sensor values will be
    evaluated for Newton invariant preservation
        # at every read. Host lang compiler will set the global variable
    INVALID if invariant not satisfied.

        x_acc = read_from_accelerometer_x()
        y_acc = read_from_accelerometer_y()
```

```
        z_acc = read_from_accelerometer_z()
        x_gyro = read_from_gyro_row()
        y_gyro = read_from_gyro_pitch()
        z_gyro = read_from_gyro_yaw()
        x_pressure = read_from_pressure_sensor()
        x_mag = read_from_magnetic_field_x()
        y_mag = read_from_magnetic_field_y()
        z_mag = read_from_magnetic_field_z()

        sample_feature_vector = np.array([
            x_acc, y_acc, z_acc, x_gyro, y_gyro,
            z_gyro, x_pres, x_mag, y_mag, z_mag
        ])

        feature_matrix.append(sample_feature_vector)

        sleep(100)

    return feature_matrix


def get_classifier(label_filename):
    feature_matrix = collect_data()
    labels = get_labels(label_filename)
    classifier = MLPClassifier()
    return classifier.fit(feature_matrix, labels)

@@error
def global_error_handler(report):
    raise Exception("An invariant is violated")
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The below program shows a transformed version of the method `collect_data`.

```
def collect_data():
    feature_matrix = []
    while read_from_system_clock() < 10000000:

        x_acc = read_from_accelerometer_x()
        y_acc = read_from_accelerometer_y()
        z_acc = read_from_accelerometer_z()
        x_gyro = read_from_gyro_row()
        y_gyro = read_from_gyro_pitch()
        z_gyro = read_from_gyro_yaw()
        x_pressure = read_from_pressure_sensor()
        x_mag = read_from_magnetic_field_x()
        y_mag = read_from_magnetic_field_y()
        z_mag = read_from_magnetic_field_z()
```

```
15
16          report = newton_api_satisfies_constraints('''parameter tree made of
        x_acc, y_acc, z_acc, x_gyro, y_gyro, z_gyro''')
17          if not report.satisfies_value_constraints:
18              global_error_handler(report)
19
20          sample_feature_vector = np.array([
21              x_acc, y_acc, z_acc, x_gyro, y_gyro,
22              z_gyro, x_pres, x_mag, y_mag, z_mag
23          ])
24
25          feature_matrix.append(sample_feature_vector)
26
27          sleep(100)
28
29      return feature_matrix
```

If the invariant is not satisfied at runtime, the transformed code above will execute the error handler function before the first usage of the variables that are parameters of the Newton invariant. To facilitate presentation, the variables that will be parameters of the Newton runtime library call are declared before used, but as mentioned in Section 6.2, the prerequisite to performing this transformation to the host language IR is that all variables must be declared before used and are defined exactly once.

Now we show the second transformation that reduces sensor redundancy. Imagine that the person collecting the sensor data wants to disable sensor redundancy intentionally to test if the classifier can categorize noisy sensor data. The transformed portion of the code is same as in the pedometer example, which is the accelerometer reader function. The original accelerometer reader function averages the output of the two redundant accelerometers.

```
1 acceleration read_from_accelerometerX() {
2     acceleration@0 average = (read_from_accelerometerX1() + (acceleration@0)
      read_from_accelerometerX2()) / 2;
3     return average;
4 }
```

After applying the transformation that reduces sensor redundancy, the above code becomes the following.

```
1 acceleration read_from_accelerometerX() {
2     acceleration@0 average = (read_from_accelerometerX1() +
      read_from_accelerometerX1()) / 2;
3     return average;
4 }
```

A.3 Maintaining Vehicle Distance

As mentioned in Section 5.5.3, this section describes a system of autonomous vehicles driving in a line, each of which tries to maintain a certain distance from the previous vehicle and respond quickly to any disturbances. Each vehicle takes a command from either the head vehicle or a remote server which calculates the speed necessary for each vehicle to reach stability of the system [9].

Sensors used by each vehicle in this code are an accelerometer, a gear shaft rotation sensor, and a sonar radar sensor (gives distance from the previous vehicle). The invariant simply states that the vehicle should maintain a reference distance from the previous vehicle.

C code

```
// https://ths.rwth-aachen.de/research/projects/hypro/n_vehicle_platoon/
// Simulate 5 autonomously driven vehicles trying to keep distance from each other

//
//                              Each individual car
//


int car_index = 3;
distance reference_distance = 1.3;

distance
read_from_radar_sensor();

distance
read_from_gear_shaft_velocity_sensor();

void
update_relative_state_vector() {
    if (car_index > 0) // if not the first car
    {
        distance distance_from_prev_car = read_from_radar_sensor();
        velocity current_velocity = read_from_gear_shaft_velocity_sensor();
        acceleration current_acceleration = read_from_accelerometer();

        relative_position = distance_from_prev_car - reference_distance;
        relative_velocity = current_velocity - ask_server_prev_car_velocity();
        relative_acceleration = current_acceleration - ask_server_prev_car_acceleration(car_index);

        send_state_to_server(
```

```
            car_index,
            relative_position,
            relative_velocity,
            relative_acceleration,
            current_acceleration,
            current_velocity
        );
    }
}

void
receive_command_from_server() {
    speed target_speed = (speed) /* parse HTTP request */
    if (read_from_from_gear_shaft_velocity_sensor() < target_speed)
    {
        speedup();
    }
    else
    {
        slowdown();
    }
}

void
send_state_to_server(
    int car_index,
    position relative_position,
    velocity relative_velocity,
    acceleration relative_acceleration,
    acceleration absolute_acceleration,
    velocity absolute_velocity
) {
    /* Send some HTTP request or signal to a server in a remote location or
    in one of the cars*/
}


//
    ********************************************************************************

//              A server controlling each car's speed
//
    ********************************************************************************

distance relative_positions[5];
velocity relative_velocities[5];
acceleration relative_accelerations[5];

acceleration absolute_accelerations[5];
velocity absolute_velocities[5];

distance reference_distance = 1.3;

void
send_prev_car_absolute_acceleration() {
    int car_index = /* parse HTTP request */
```

```
81      /* send absolute_accelerations[car_index-1] to vehicle number car_index -
        1 */
82  }
83
84  void
85  send_prev_car_absolute_velocity() {
86      int car_index = /* parse HTTP request */
87      /* send absolute_velocities[car_index-1] to vehicle number car_index - 1
        */
88  }
89
90  void
91  receive_state_from_client_vehicle()
92  {
93      /* parse HTTP request parameters */
94
95      relative_positions[car_index] = relative_position;
96      relative_velocities[car_index] = relative_velocity;
97      relative_accelerations[car_index] = relative_acceleration;
98
99      absolute_accelerations[car_index] = current_acceleration;
100     absolute_velocities[car_index] = current_velocity;
101 }
102
103 void
104 solve_controls_and_send_commands()
105 {
106     /* Compute closed loop system by solving H2 optimization problem and then
        command vehicles how to react */
107 }
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `update_relative_state_vector` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
1  void
2  update_relative_state_vector(int car_index) {
3      if (car_index > 0) // if not the first car
4      {
5          distance distance_from_prev_car = read_from_radar_sensor();
6          velocity current_velocity = read_from_gear_shaft_velocity_sensor();
7          acceleration current_acceleration = read_from_accelerometer();
8
9          NewtonAPIReport* report = newtonApiSatisfiesConstraints(
10             newton,
```

```
            /* parameter tree made of distance_from_prev_car */
        );
        VALID = VALID && report->satisfiesValueConstraint;
        if (!VALID)
            global_error_handler(report);

        NewtonAPIReport* report = newtonApiSatisfiesConstraints(
            newton,
            /* parameter tree made of distance_from_prev_car and
    current_velocity */
        );
        VALID = VALID && report->satisfiesValueConstraint;
        if (!VALID)
            global_error_handler(report);

        relative_position = distance_from_prev_car - reference_distance;
        relative_velocity = current_velocity - ask_server_prev_car_velocity()
    ;
        relative_acceleration = current_acceleration -
    ask_server_prev_car_acceleration(car_index);

        send_state_to_server(
            car_index,
            relative_position,
            relative_velocity,
            relative_acceleration,
            current_acceleration,
            current_velocity
        );
    }
}
```

A.4 Weather Balloon

As mentioned in Section 5.5.4, this system is a weather balloon with various sensors including temperature sensors, altitude sensor, pressure sensor, and humidity sensor that simply floats in the air and records sensor data. This example focuses on temperature and pressure sensor data. The invariant is that the atmospheric temperature and the pressure are related by the gas laws that describe the International Standard Atmosphere [10].

The C program for the weather balloon written below describes a weather balloon reading data from sensors and writing them to a file.

C code

```
void record_weather_conditions() {
    char* weather_file = "/path/to/weather.txt";
    while (is_rising()) {
```

```
        pressure ground_pressure = read_from_pressure_sensor();
        record_pressure(weather_file, ground_pressure);

        temperature ground_temp = read_from_temperature_sensor();
        record_temperature(weather_file, ground_temp);

        sleep(1000);
    }
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    ConstraintReport* currentConstraint = report->firstConstraintReport;
    while (currentConstraint != NULL) {
        /* log errors in some file */
        currentConstraint = firstConstraint->next;
    }
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `record_weather_conditions` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

C code

```
void record_weather_conditions()
{
    char* weather_file = "/path/to/weather.txt";
    while (is_rising()) {
        bool VALID = true;

        pressure air_pressure = read_from_pressure_sensor();
        record_pressure(weather_file, air_pressure);

        temperature air_temp = read_from_temperature_sensor();

        NewtonAPIReport* report = newtonApiSatisfiesConstraints(
            newton,
            /* parameter tree made of air_pressure and air_temp */
        );
        VALID = VALID && report->satisfiesValueConstraint;
        if (!VALID)
            global_error_handler(report);

        record_temperature(weather_file, air_temp);
```

```
21
22          sleep(1000);
23      }
24  }
```

The Newton runtime library call `newtonApiSatisfiesConstraints` is called after `air_pressure` and `air_temp` have been assigned a value. At compile-time by matching types and dynamic tagging, the C compiler can find out that `air_pressure` and `air_temp` are parameters to the Newton invariant. Then, it finds a point, after all parameters have been assigned, that is before any parameter has been used. In this code, after both parameters have been assigned a value, `air_pressure` is already used. However, if the invariant was found to be violated, the global error handler will execute before the usage of `air_temp`.

A.5 GPS Walking

The application here is an example from [15] where a person is walking with a smart-watch with an accelerometer and a GPS. The person receives an alert from the smart-watch if she is walking below a target speed. The Newton description for this application is in Section 5.5.5

The invariants are that the speed computed from the GPS location simply by dividing by sampling time should roughly match the speed computed from the accelerometers in the sensors by multiplying by sampling time.

The following code uses annotations to designate some functions as routines that return values meant to be used for invariant checking. In the code below, the annotated functions are the accelerometer reader functions, `get_distance_moved`, and `get_speed`. This annotation allows the host language compiler to identify variables whose values are assigned directly from those functions and mark them as a potential parameter to be checked for an invariant. See Section 6.2.2 for more information.

Note here that this code assumes that type casting feature when it is adding variables of distance types. The current implementation does not allow adding variables of different axes because different sub-dimensions are treated as separate Physics types. This can be resolved by allowing type casting on individual Physics expressions.

```
// Uncertain<T> by Bornholt paper. Figure 5a in the paper

void
gps_walking() {
    time dt = 50;

    distance@0 loc_x = gps.get_latitude();
    distance@1 loc_y = gps.get_longitude();

    while (true) {
        sleep(dt);
        distance@0 cur_loc_x = gps.get_latitude();
        distance@1 cur_loc_y = gps.get_longitude();

        distance@0 ds = get_distance_moved(cur_loc_x, loc_x, cur_loc_y, loc_y
    );
        speed v = get_speed(ds, dt);
        if (v > 4) GoodJob();
        else SpeedUp();

        loc_x = cur_loc_x;
        loc_y = cur_loc_y;
    }
}

@sensor
distance@0 get_distance_moved(
    distance@0 prev_loc_x,
    distance@0 cur_loc_x,
    distance@1 prev_loc_y,
    distance@1 cur_loc_y,
) {
    return (distance@3)(cur_loc_x - prev_loc_x) + (distance@3)(cur_loc_y -
    prev_loc_y);
}

@sensor
distance@0 get_speed(
    distance@3 ds,
    time dt
) {
    return ds / dt;
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    ConstraintReport* currentConstraint = report->firstConstraintReport;
    while (currentConstraint != NULL) {
        /* log errors in some file */
        currentConstraint = firstConstraint->next;
    }
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host

language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `gps_walking` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
// Uncertain<T> by Bornholt paper. Figure 5a in the paper

void
gps_walking() {
    time dt = 50;

    distance@0 loc_x = gps.get_latitude();
    distance@1 loc_y = gps.get_longitude();

    while (true) {
        sleep(dt);
        distance@0 cur_loc_x = gps.get_latitude();
        distance@1 cur_loc_y = gps.get_longitude();

        distance@3 ds = get_distance_moved(cur_loc_x, loc_x, cur_loc_y, loc_y
    );
        speed v = get_speed(ds, dt);

        NewtonAPIReport* report = newtonApiSatisfiesConstraints(
            newton,
            /* parameter tree made of air_pressure and air_temp */
        );
        VALID = VALID && report->satisfiesValueConstraint;
        if (!VALID)
            global_error_handler(report);

        if (v > 4) GoodJob();
        else SpeedUp();

        loc_x = cur_loc_x;
        loc_y = cur_loc_y;
    }
}
```

Now we show the second transformation that reduces sensor redundancy. Suppose that the smart-watch used for GPS-Walking has an option for a low-power GPS which reduces GPS redundancy. Almost identical to the pedometer and the activity classifier example before, this transformation would affect the GPS reader function.

```
distance@0 read_from_gps_x_coordinate() {
    distance@0 average = (read_from_gps_x_coordinate1() + (distance@0)
    read_from_gps_x_coordinate2()) / 2;
    return average;
```

```
}
```

After applying the transformation that reduces sensor redundancy, the above code becomes the following.

```
distance@0 read_from_gps_x_coordinate() {
    distance@0 average = (read_from_gps_x_coordinate1() +
    read_from_gps_x_coordinate2()) / 2;
    return average;
}
```

A.6 SensorLife

This code describes the system from Section 5.5.6. This SensorLife example is identical to Conway's Game of Life except living and dying are determined by the number of neighbors over a threshold temperature. If a cell receives a signal to die, the cooling unit within the cell quickly cools the cell down to below the threshold temperature, and likewise, if a cell receives a signal to be revived, the heating unit within the cell heats up the cell. The function that transitions the grids is shown below.

```
temperature threshold = 50;

void update_one_cell(
    uint8_t cur_x,
    uint8_t cur_y,
    Cell* [max_width][max_height] current_grid,
    Cell* [max_width][max_height] next_grid
) {
    int num_live_neighbors = 0

    temperature@0 nw = read_neighbor_temp_nw(current_grid[cur_x][cur_y]);
    temperature@1 n = read_neighbor_temp_n(current_grid[cur_x][cur_y]);
    temperature@2 ne = read_neighbor_temp_ne(current_grid[cur_x][cur_y]);
    temperature@3 e = read_neighbor_temp_e(current_grid[cur_x][cur_y]);
    temperature@4 w = read_neighbor_temp_w(current_grid[cur_x][cur_y]);
    temperature@5 sw = read_neighbor_temp_sw(current_grid[cur_x][cur_y]);
    temperature@6 s = read_neighbor_temp_s(current_grid[cur_x][cur_y]);
    temperature@7 se = read_neighbor_temp_se(current_grid[cur_x][cur_y]);

    if (nw > threshold)
        num_live_neighbors++;
    if (n > threshold)
        num_live_neighbors++;
    if (ne > threshold)
        num_live_neighbors++;
    if (e > threshold)
        num_live_neighbors++;
```

```
28      if (w > threshold)
29          num_live_neighbors++;
30      if (sw > threshold)
31          num_live_neighbors++;
32      if (s > threshold)
33          num_live_neighbors++;
34      if (se > threshold)
35          num_live_neighbors++;
36
37      if (current_grid[cur_x][cur_y].alive)
38      {
39          if (num_live_neighbors < 2 || num_live_neighbors > 3)
40              kill(next_grid[cur_x][cur_y]); // cool the cell to below
        threshold
41      }
42      else
43      {
44          if (num_live_neighbors == 3)
45              revive(next_grid[cur_x][cur_y]); // heat up the cell to above
        threshold
46      }
47  }
48
49  @@error
50  void global_error_handler(NewtonAPIReport* report) {
51      ConstraintReport* currentConstraint = report->firstConstraintReport;
52      while (currentConstraint != NULL) {
53          /* log errors in some file */
54          currentConstraint = firstConstraint->next;
55      }
56  }
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `update_one_cell` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
1  temperature threshold = 50;
2
3  void update_one_cell(
4      uint8_t cur_x,
5      uint8_t cur_y,
6      Cell* [max_width][max_height] current_grid,
7      Cell* [max_width][max_height] next_grid
8  ) {
9      bool VALID = true;
```

```
10      int num_live_neighbors = 0
11
12      temperature@0 nw = read_neighbor_temp_nw(current_grid[cur_x][cur_y]);
13      temperature@1 n = read_neighbor_temp_n(current_grid[cur_x][cur_y]);
14      temperature@2 ne = read_neighbor_temp_ne(current_grid[cur_x][cur_y]);
15      temperature@3 e = read_neighbor_temp_e(current_grid[cur_x][cur_y]);
16      temperature@4 w = read_neighbor_temp_w(current_grid[cur_x][cur_y]);
17      temperature@5 sw = read_neighbor_temp_sw(current_grid[cur_x][cur_y]);
18      temperature@6 s = read_neighbor_temp_s(current_grid[cur_x][cur_y]);
19      temperature@7 se = read_neighbor_temp_se(current_grid[cur_x][cur_y]);
20
21      NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
        parameter tree made of above variables */);
22      VALID = VALID && report->satisfiesValueConstraint;
23      if (!VALID)
24          global_error_handler(report);
25
26      if (nw > threshold)
27          num_live_neighbors++;
28      if (n > threshold)
29          num_live_neighbors++;
30      if (ne > threshold)
31          num_live_neighbors++;
32      if (e > threshold)
33          num_live_neighbors++;
34      if (w > threshold)
35          num_live_neighbors++;
36      if (sw > threshold)
37          num_live_neighbors++;
38      if (s > threshold)
39          num_live_neighbors++;
40      if (se > threshold)
41          num_live_neighbors++;
42
43      if (current_grid[cur_x][cur_y].alive)
44      {
45          if (num_live_neighbors < 2 || num_live_neighbors > 3)
46              kill(next_grid[cur_x][cur_y]); // cool the cell to below
        threshold
47      }
48      else
49      {
50          if (num_live_neighbors == 3)
51              revive(next_grid[cur_x][cur_y]); // heat up the cell to above
        threshold
52      }
53  }
```

Now suppose that we want to apply the sensor redundancy reducing transformation to this code. As in the pedometer, the activity classifier, and GPS-Walking examples, this transformation concerns the temperature sensor redundancy. In this example, a total of 9 temperature sensor usages would be reduced because there are two redundant temperature

sensors for each of the 9 sets of sensors. The code below shows a reader function for one of the temperature sensors. The transformation below would be repeated for the other 8 sets of sensors as well.

```
temperature@0 read_from_gps_x_coordinate() {
    temperature@0 average = (read_from_temperature_sensor_0() + (
    temperature@9) read_from_temperature_sensor_9()) / 2;
    return average;
}
```

After applying the transformation that reduces sensor redundancy, the above code becomes the following.

```
temperature@0 read_from_gps_x_coordinate() {
    temperature@0 average = (read_from_temperature_sensor_0() +
    read_from_temperature_sensor_0()) / 2;
    return average;
}
```

A.7 Ball Dropped from a Height

This C code as mentioned in Section 5.5.7 describes a simple Physics experiment where sensors are placed inside a ball and various sensor data are measured.

The invariant is that the ball should lose mechanical energy as it bounces against the ground.

```
/*
 * Just imagine observing a sensor ball dropped from 10 m above the ground
    and measuring
 * how it behaves. The ball has an altitude sensor and 3 axes accelerometer
 */
void record_measurements()
{
    while (true)
    {
        acceleration@0 x = read_from_accelerometer_x();
        acceleration@1 y = read_from_accelerometer_y();
        acceleration@2 z = read_from_accelerometer_z();
        distance@2 altitude = read_from_altitude_sensor();

        record_acceleration_x(x);
        record_acceleration_y(y);
        record_acceleration_z(z);
        record_altitude(altitude);
    }
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `record_measurements` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
1  void record_measurements()
2  {
3      while (true)
4      {
5          acceleration@0 x = read_from_accelerometer_x();
6          acceleration@1 y = read_from_accelerometer_y();
7          acceleration@2 z = read_from_accelerometer_z();
8          distance@2 altitude = read_from_altitude_sensor();
9
10         NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
       parameter tree made of above variables */);
11         VALID = VALID && report->satisfiesValueConstraint;
12         if (!VALID)
13             global_error_handler(report);
14
15         record_acceleration_x(x);
16         record_acceleration_y(y);
17         record_acceleration_z(z);
18         record_altitude(altitude);
19     }
20 }
```

A.8 Jet Engine

The following code models what the software monitoring a jet engine would do assuming that the engine follows the Moor Greitzer Jet Engine Model. The code simply monitors the conditions of the jet engine. The Newton description for this code is in Section 5.5.8.

```
1  void monitor_jet_engine()
2  {
3      mass_flow_rate flow = read_from_mass_flow_rate_sensor();
4      pressure engine_pressure = read_from_pressure_sensor();
5      ...
6      /* there might be more sensors here */
7
8      display_info_monitor();
```

```
}

@@error
global_error_handler(NewtonAPIReport* report) {
    /* display errors to the pilot on a monitor */
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `monitor_jet_engine` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
void monitor_jet_engine()
{
    mass_flow_rate flow = read_from_mass_flow_rate_sensor();
    pressure engine_pressure = read_from_pressure_sensor();

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of above variables */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    ...
    /* there might be more sensors here */

    display_info_monitor();
}
```

A.9 Reactor Rod Cooling

This section describes a monitoring application of reactor rod temperature. Reactor rods are usually submerged under some coolants like heavy water. The rods cool off faster if the water has better cooling coefficients. The following code describes a monitoring application with a Newton invariant. The invariant is that the time it would take for the rod to cool off to a target temperature if the power was off should not exceed the maximum amount of time allowed for the reactor to be turned off. See the Newton description in Section 5.5.9.

```
/*
 * The goal is to control the temperature of the rods by swapping out
   coolants when rod temperature rises.
 */

void monitor_reactor_temperature()
{
    temperature@0 rod_temp = read_from_rod_temperature_sensor();
    temperature@0 coolant_temp = read_from_coolant_temperature_sensor();
    if (rod_temp > 300 && coolant_temp > 50)
    {
        change_coolant();
    }
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    alert_operator("Reactor rod temperature very high");
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `monitor_reactor_temperature` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied. The error handler here would alert the reactor operator who would then take necessary steps to control the temperature.

```
void monitor_reactor_temperature()
{
    temperature@0 rod_temp = read_from_rod_temperature_sensor();
    temperature@0 coolant_temp = read_from_coolant_temperature_sensor();

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of above variables */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    if (rod_temp > 300 && coolant_temp > 50)
    {
        change_coolant();
    }
}
```

A.10 Airplane Altitude and Speed

As explained in Section 5.5.10, airplanes have two pressure sensors in its altimeter and its pitot tube. It is essential for those pressure sensors to report identical readings, or the altitude and the velocity readings would be incorrect. The following code represents what an autopilot software would do to monitor the aircraft conditions and determine the air speed.

```
void aircraft_speed_controller()
{
    speed aircraft_speed = read_from_speed_sensor();
    distance@2 altitude = read_from_altimeter();

    if (altitude < 3000 && aircraft_speed < 1000)
    {
        speed_up();
    }
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    alert_pilot("Pressure sensors in altimeter and pitot tube are broken.");
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `aircraft_speed_controller` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied. The error handler here would alert the reactor operator who would then take necessary steps to control the temperature.

```
void aircraft_speed_controller()
{
    speed aircraft_speed = read_from_speed_sensor();
    distance@2 altitude = read_from_altimeter();

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of above variables */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    if (altitude < 3000 && aircraft_speed < 1000)
    {
```

```
        speed_up();
    }
}
```

A.11 Motorized Wheel Chair

This section describes a programmable motorized wheel chair mentioned in Section 5.5.11. Imagine that the programmer wants the wheel chair to be automated and that the motor of the wheel chair should only turn on if there is someone sitting on the chair for safety. The following code reads from the sensors available on the wheel chair to see if they are still working before moving. Newton invariants are used in `check_sensors` to check additional constraints specified by the hardware manufacturer.

```
/*
 * a motor wheel chair should turn on only if there is
 * someone sitting on it.
 */

void start_moving(int direction) {
    check_sensors();
    speed_up(direction);
}

void check_sensors() {
    pressure seat_pressure = read_from_pressure_sensor();
    temperature seat_temp = read_from_temperature_sensor();
    acceleration@0 x = read_from_accelerometer_x();
    acceleration@1 y = read_from_accelerometer_y();
    acceleration@2 z = read_from_accelerometer_z();

    assert(seat_pressure != 0);
    assert(seat_temp != 0);
    assert(x != 0);
    assert(y != 0);
    assert(z != 0);
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    display("Safety check failed");
    slowdown_to_halt();
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton

runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `check_sensors` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied. The error handler function lets the rider know that the safety check has failed and slows down the wheel chair to a stop.

```
void check_sensors() {
    pressure seat_pressure = read_from_pressure_sensor();
    temperature seat_temp = read_from_temperature_sensor();
    acceleration@0 x = read_from_accelerometer_x();
    acceleration@1 y = read_from_accelerometer_y();
    acceleration@2 z = read_from_accelerometer_z();

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of seat_pressure and seat_temp */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of x, y, and z */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);

    assert(seat_pressure != 0);
    assert(seat_temp != 0);
    assert(x != 0);
    assert(y != 0);
    assert(z != 0);
}
```

A.12 Car Tire Pressure And Acceleration Range

This application is a software that monitors conditions of a car by reading sensor values and displaying them on a dashboard monitor display. The invariant is simply a value constraint on tire pressure and car acceleration.

```
// pressure sensors in car tires
// https://www.kaltire.com/the-right-tire-pressure-why-the-maximum-isnt-the-
    best/

void
display_car_condition_on_dashboard() {
    pressure tire_pressure = read_from_tire_pressure_sensor();
    float remaining_gas_in_gallons = read_from_gas_meter();
```

```
    speed current_speed = read_from_speedometer();
    ...

    /* display the above info on monitor */
}

@@error
void global_error_handler(NewtonAPIReport* report) {
    display("Tire pressure abnormal");
}
```

There needs to be a series of Newton API calls, as mentioned in Chapter 4, to perform dimensional type checking, construct Newton AST of the source code, and select host language program variables as parameters to Newton invariants before inserting the Newton runtime library call `newtonApiSatisfiesConstraints` into the host language IR.

Read Chapter 4 for a more detailed step-by-step explanation of a similar example.

The code below is a transformed version of the method `display_car_condition_on_dashboard` that checks invariants at runtime and executes the global error handler function if the invariant is not satisfied.

```
void
display_car_condition_on_dashboard() {
    pressure tire_pressure = read_from_tire_pressure_sensor();
    float remaining_gas_in_gallons = read_from_gas_meter();
    speed current_speed = read_from_speedometer();
    ...

    NewtonAPIReport* report = newtonApiSatisfiesConstraints(newton, /*
    parameter tree made of tire_pressure */);
    VALID = VALID && report->satisfiesValueConstraint;
    if (!VALID)
        global_error_handler(report);


    /* display the above info on monitor */
}
```

Appendix B

Appendix: Formal Grammar for the Newton Description File

```
<Newton-file> ::= <rule-list>

<rule-list> ::= [<rule>]+
<rule> ::=  (<constant> | <invariant> | <base-signal>)

<constant> ::= <identifier> ':' 'constant' '=' <number> <unit-expression>
<invariant> ::= <identifier> ':' 'invariant' <parameter-tuple>  '=' '{' <
    constraint-list> '}'
<base-signal> ::= <identifier> ':' 'signal' [subdimension-tuple]? '=' '{' <
    name> <symbol> <derivation> '}'

<name> ::= 'name' '=' <string-constant> <language-setting>
<symbol> ::= 'symbol' '=' <string-constant> ';'
<derivation> ::= 'derivation' '=' ('none' | <quantity-expression>) ';'

<subdimension-tuple> ::= '(' <identifier> ':' <number> 'to' <number> ')'

<parameter-tuple> ::= '(' <parameter> [',' <parameter>]* ')'
<parameter> ::= <identifier> ':' <identifier>

<constraint-list> ::= <constraint> [',' <constraint>]*
<constraint> ::= [
                    <quantity-expression> <compare-op> <quantity-expression>
    |
                    <quantity> <assign-op> <quantity-expression>
                ]

<quantity-expression> ::= <quantity-term> [<low-precedence-op> <quantity-term
    >]*
<quantity-term> ::= ['-']? <quantity-factor> [<mid-precedence-op> <quantity-
    factor>]*
<quantity-factor> ::=  [
                        <quantity> [<high-precedence-op> <quantity-
```

```
    expression>]? |
                          <time-op>* <quantity-expression> |
                          '(' <quantity-expression> ')' |
                          <vector-op> '(' <quantity-expression> ',' <quantity-
    expression>')'
                        ]
<quantity> ::= <number> | (<identifier> ['@' <number>]*)
<low-precedence-op> ::= '+' | '-'
<mid-precedence-op> ::= '*' | '/'
<high-precedence-op> ::= '**'
<vector-op> ::= 'dot' | 'cross'
<time-op> ::= 'derivative' | 'integral'
<assign-op> ::= '='

<unit-expression> ::= <unit-term>
<unit-term> ::= <unit-factor> [<mid-precedence-op> <unit-factor>]+
<unit-factor> ::= ['-']? (<unit> [<high-precedence-op> <number>]? | '(' <unit
    -expression> ')')
<unit> ::= <identifier>

<number> ::= ['-']? <integer> ['.' <integer>]*
<integer> ::= [1..9][0..9]*
<string-constant> ::= '"'[a-zA-Z]+'"'
<identifier> ::= [a-zA-Z][0-9a-zA-Z\_]*
<compare-op> ::= ['o<' | '~' | '<' | '<=' | '>' | '>=']
<language-setting> ::= ('English' | 'Spanish')
```

Appendix C

Appendix: Data Structures Used

```
/*
 * Checking on each constraint will add a ConstraintReport struct
 * to the NewtonAPIReport struct. While walking the expression tree,
 * the Newton API compares the dimensions and the values of the operands
 * of a binary operation and generates a report.
 */
struct ConstraintReport
{
  bool satisfiesDimensionConstraint;
  char dimensionErrorMessage[1024];
  bool satisfiesValueConstraint;
  char valueErrorMessage[1024];

  SourceInfo * failedLocation;

  ConstraintReport* next;
};

/*
 * Contains a linked list of ConstraintReport's
 * Each call to newtonApiSatisfiesConstraints will generate a NewtonAPIReport
 */
struct NewtonAPIReport
{
  ConstraintReport* firstConstraintReport;
};

typedef struct Scope  Scope;
typedef struct Symbol Symbol;
typedef struct Token  Token;
typedef struct IrNode IrNode;
typedef struct SourceInfo SourceInfo;
typedef struct Dimension Dimension;
typedef struct Physics Physics;
typedef struct IntegralList IntegralList
;
typedef struct Invariant Invariant;
```

```
38
39 struct Dimension
40 {
41     char * identifier;
42     char * abbreviation;
43
44     double exponent; // default value is 1 if exists
45
46     Scope *   scope;
47
48     SourceInfo *  sourceInfo;
49
50
51     int primeNumber;
52
53     Dimension * next;
54 };
55
56 struct Invariant
57 {
58     char * identifier; // name of the physics quantity. of type
       kNoisyConfigType_Tidentifier
59
60
61
62     Scope *   scope;
63     SourceInfo *  sourceInfo;
64
65
66
67     IrNode * parameterList; // this is just bunch of IrNode's in Xseq
68     unsigned long long int id;
69
70     IrNode * constraints;
71
72     Invariant * next;
73 };
74
75 struct Physics
76 {
77     char * identifier; // name of the physics quantity. of type
       kNoisyConfigType_Tidentifier
78     unsigned long long int id;
79   int subindex; /* index for further identification. e.g.) acceleration along
       x, y, z axes */
80
81     Scope *   scope;
82     SourceInfo *  sourceInfo;
83
84     double value; /* for constants like Pi or gravitational acceleration */
85     bool isConstant;
86
87     Dimension * dimensions;
88
89     char * dimensionAlias;
90     char * dimensionAliasAbbreviation;
```

```
91
92      Physics * definition;
93
94      Physics * next;
95  };
96
97  struct IrNode
98  {
99    IrNodeType    type;
100
101   /*
102    *  Syntactic (AST) information.
103    */
104   char *      tokenString;
105   Token * token;
106   SourceInfo  * sourceInfo;
107   IrNode *    irParent;
108   IrNode *    irLeftChild;
109   IrNode *    irRightChild;
110
111   Symbol *    symbol;
112
113
114
115   /*
116    * Used for evaluating dimensions in expressions
117    */
118   Physics * physics;
119
120   /*
121    * only if this node belongs to a ParseNumericExpression subtree
122    */
123   double value;
124
125   int subindexStart;
126   int subindexEnd;
127
128   /*
129    * A parameter tuple of length n has ordering from zero to n - 1
130    */
131   int parameterNumber;
132
133   /*
134    * When doing an API check of the invariant tree given a parameter tree,
135    * the method looks up all instances of
136    */
137
138   /*
139    *  Used for coloring the IR tree, e.g., during Graphviz/dot generation
140    */
141   IrNodeColor nodeColor;
142 };
143
144
145 struct SourceInfo
146 {
```

```
147   char **     genealogy;
148
149   char *      fileName;
150   uint64_t    lineNumber;
151   uint64_t    columnNumber;
152   uint64_t    length;
153 };
154
155
156 struct Token
157 {
158   IrNodeType    type;
159   char *      identifier;
160   uint64_t    integerConst;
161   double      realConst;
162   char *      stringConst;
163   SourceInfo *  sourceInfo;
164
165   Token *   prev;
166   Token *   next;
167 };
168
169
170 struct Scope
171 {
172   /*
173    *  For named scopes (at the moment, only Progtypes)
174    */
175   char *      identifier;
176
177    int currentSubindex;
178
179   /*
180    *  Hierarchy. The firstChild is used to access its siblings via firstChild
       ->next
181    */
182   Scope *   parent;
183   Scope *   firstChild;
184
185   /*
186    *  Symbols in this scope. The list of symbols is accesed via firstSymbol->
       next
187    */
188   Symbol *    firstSymbol;
189
190    /*
191     * each invariant scope will have its own list of parameters
192     */
193    IrNode * invariantParameterList; // this is just bunch of IrNode's in
       Xseq
194
195    /*
196     * For the config file, we only have one global scope that keeps track of
       all
197     * dimensions and Physics types.
198     */
```

```
199     Dimension * firstDimension;
200     Physics * firstPhysics;
201
202   /*
203    *  Where in source scope begins and ends
204    */
205   SourceInfo *  begin;
206   SourceInfo *  end;
207
208   /*
209    *  For chaining together scopes (currently only used for Progtype
210    *  scopes and for chaining together children).
211    */
212   Scope *   next;
213   Scope *   prev;
214
215   /*
216    *  Used for coloring the IR tree, e.g., during Graphviz/dot generation
217    */
218   IrNodeColor nodeColor;
219 };
220
221
222 struct Symbol
223 {
224   char *      identifier;
225
226   /*
227    *  This field is duplicated in the AST node, since only
228    *  identifiers get into the symbol table:
229    */
230   SourceInfo *  sourceInfo;
231
232   /*
233    *  Declaration, type definition, use, etc. (kNoisySymbolTypeXXX)
234    */
235   int       symbolType;
236
237   /*
238    *  Scope within which sym appears
239    */
240   Scope *   scope;
241
242   /*
243    *  If an identifier use, definition's Sym, if any
244    */
245   Symbol *    definition;
246
247   /*
248    *  Subtree in AST that represents typeexpr
249    */
250   IrNode *    typeTree;
251
252   /*
253    *  If an I_CONST, its value.
254    */
```

```
255   int     intConst;
256   double      realConst;
257   char *      stringConst;
258
259   /*
260    *  For chaining together sibling symbols in the same scope
261    */
262   Symbol *    next;
263   Symbol *    prev;
264 };
265
266
267 typedef struct
268 {
269   /*
270    *  Timestamps to track lifecycle
271    */
272   uint64_t    initializationTimestamp;
273   TimeStamp * timestamps;
274   uint64_t    timestampCount;
275   uint64_t    timestampSlots;
276
277
278   /*
279    *  Track aggregate time spent in all routines, by incrementing
280    *  timeAggregates[timeAggregatesLastKey] by (now -
       timeAggregatesLastTimestamp)
281    */
282   uint64_t *    timeAggregates;
283   TimeStampKey  timeAggregatesLastKey;
284   uint64_t    timeAggregatesLastTimestamp;
285   uint64_t    timeAggregateTotal;
286   uint64_t *    callAggregates;
287   uint64_t    callAggregateTotal;
288
289
290   /*
291    *  Used to get error status from FlexLib routines
292    */
293   FlexErrState *    Fe;
294
295   /*
296    *  State for the portable/monitoring allocator (FlexM)
297    */
298   FlexMstate *    Fm;
299
300   /*
301    *  State for portable/buffering print routines (FlexP)
302    *  We have one buffer for informational messages, another
303    *  for errors and warnings.
304    */
305   FlexPrintBuf *    Fperr;
306   FlexPrintBuf *    Fpinfo;
307
308
309   /*
```

```
310     *  The output file of the last render. TODO: Not very happy
311     *  with this solution as it stands... (inherited from Sal/svm)
312     */
313    char *      lastDotRender;
314
315
316    /*
317     *  This is the name of the progtype that the file we're parsing implements
318     */
319    char *      progtypeOfFile;
320
321    /*
322     *  We keep a global handle on the list of progtype scopes, for easy
        reference.
323     *  In this use case, the node->identifier holds the scopes string name,
        and we
324     *  chain then using their prev/next fields.
325     */
326    Scope *   progtypeScopes;
327
328
329    /*
330     *  Lexer state
331     */
332    char *      fileName;
333    char *      lineBuffer;
334    uint64_t    columnNumber;
335    uint64_t    lineNumber;
336    uint64_t    lineLength;
337    char *      currentToken;
338    uint64_t    currentTokenLength;
339    Token *   tokenList;
340    Token *   lastToken;
341
342
343    /*
344     *  The root of the IR tree, and top scope
345     */
346    IrNode *    noisyIrRoot;
347    IrNode *    newtonIrRoot;
348    Scope *   noisyIrTopScope;
349      Scope *        newtonIrTopScope;
350
351    /*
352     *  Output file name when emitting bytecode/protobuf
353     */
354    char *      outputFilePath;
355
356    NoisyMode   mode;
357    uint64_t    verbosityLevel;
358    uint64_t    dotDetailLevel;
359    uint64_t    optimizationLevel;
360    uint64_t    irPasses;
361    uint64_t    irBackends;
362
363
```

```
364   jmp_buf     jmpbuf;
365   bool      jmpbufIsValid;
366
367     /*
368      * Global index of which prime numbers we have used for the dimension id'
      s
369      */
370     int primeNumbersIndex;
371
372   /*
373    * When parsing invariant constraints, need to number the factors that
       correspond to the parameters passed in.
374    * This is so that finding matching Parameter doesn't depend either the
       identifier passed, or the physics type.
375    * That is a good idea because now we don't need to implicitly fill in the
       left identifier child of the parameter node.
376    */
377   int currentParameterNumber;
378
379   Invariant * invariantList;
380
381 } State;
```

Appendix D

Appendix: The Header File for the Newton API

```
1   /*
2    * Given a path to a Newton description file
3    * return a global state that contains the Newton IR and the symbol
    tables.
4    * The symbol table contains the list of all Physics structs and
    Dimension structs.
5    * This state is used to make all other API calls.
6    */
7   NoisyState * newtonApiInit(char *  newtonFileName);
8
9   /*
10   * Given a Newton state and a string, get a Physics type with the name of
     that string.
11   * return NULL if there isn't a Physics struct with the given name in the
     Newton description
12   * passed into the API call newtonApiInit.
13   */
14  Physics* newtonApiGetPhysicsTypeByName(NoisyState* N, char* nameOfType);
15
16  /*
17   * Given a Newton state and a tree of kNewtonIrNodeType_Pparameter nodes,
     which have
18   * left child with kNewtonIrNodeType_Tidentifier describing the name of
    the variable of a Physics type
19   * and a numeric value assigned to that variable
20   * and a right child with kNewtonIrNodeType_Tidentifier describing the
    name of the Physics type and containing
21   * in its physics field the result of the API call
    newtonApiGetPhysicsTypeByName(newton, physicsString),
22   * return a NewtonAPIReport struct that constains information about
    whether the constraints between
23   * different Physics structs are satisfied.
24   */
```

```
25  NewtonAPIReport* newtonApiSatisfiesConstraints(NoisyState* N, NoisyIrNode
    * parameterTreeRoot);
26
27  /*
28   * Given a Newton state and a tree of kNewtonIrNodeType_Pparameter nodes,
29   * return an Invariant which contains constraints made of the variables
    with Physics types
30   * mentioned in the given parameter tree.
31   */
32  Invariant * newtonApiGetInvariantByParameters(NoisyState* N, NoisyIrNode*
     parameterTreeRoot);
33
34
35  /*
36   * Makes an Ir node.
37   */
38  genNoisyIrNode(NoisyState* N, NoisyIrNodeType type, NoisyIrNode* left,
    NoisyIrNode* right, NoisySourceInfo* source);
39
40  /*
41   * Makes an Ir node and set its value.
42   */
43  makeNoisyIrNodeSetToken(NoisyState* N, NoisyIrNodeType type, char*
    identifier, char* stringConst, double numericValue);
44
45  /*
46   * Adds a node to tree in post-order traversal manner.
47   */
48  addLeaf(NoisyState* N, NoisyIrNode* root, NoisyIrNode* child);
49
50  /*
51   * Adds a node to tree in post-order traversal manner but add a Xseq node
     before attaching the child
52   */
53  addLeafWithChainingSeqNoLexer(NoisyState* N, NoisyIrNode* root,
    NoisyIrNode* child);
```

Bibliography

[1] Aeroperu flight 603. https://en.wikipedia.org/wiki/Aeroper%C3%BA_Flight_603. Accessed: 2017-05-16.

[2] Airplane was flying below recommended landing speed before crash. https://ww2.kqed.org/news/2013/07/07/102813/investigation-sfo-flight-214-crash/. Accessed: 2017-05-13.

[3] Conway's game of life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life. Accessed: 2017-05-16.

[4] Duck typing. https://en.wikipedia.org/wiki/Duck_typing. Accessed: 2017-05-13.

[5] Electrical sensors. http://www.toptransmissions.com/electrical-sensors. Accessed: 2017-05-13.

[6] Fail-fast. https://martinfowler.com/ieeeSoftware/failFast.pdf. Accessed: 2017-06-08.

[7] Full-featured pedometer design realized with 3-axis digital accelerometer. http://www.analog.com/media/en/technical-documentation/technical-articles/pedometer.pdf. Accessed: 2017-06-15.

[8] Google io. https://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf. Accessed: 2017-06-04.

[9] H2-based n-vehicle platoon. https://ths.rwth-aachen.de/research/projects/hypro/n_vehicle_platoon/. Accessed: 2017-06-18.

[10] The international standard atmosphere. http://home.anadolu.edu.tr/~mcavcar/common/ISAweb.pdf. Accessed: 2017-05-16.

[11] The physics of usain bolt's world record 100-meter dash. http://io9.gizmodo.com/the-physics-of-usain-bolts-world-record-100-meter-dash-924744818. Accessed: 2017-06-15.

[12] Stability and robustness analysis of nonlinear systems via contractionmetrics and sos programming. http://web.mit.edu/nsl/www/preprints/SOS_metrics.pdf. Accessed: 2017-06-18.

[13] typing. https://docs.python.org/3/library/typing.html. Accessed: 2017-06-18.

[14] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[15] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain: A first-order type for uncertain data. *SIGPLAN Not.*, 49(4):51–66, Feb. 2014.

[16] L. Brand. The pi theorem of dimensional analysis. *Archive for Rational Mechanics and Analysis*, 1(1):35–45, 1957.

[17] E. Buckingham. On physically similar systems; illustrations of the use of dimensional equations. *Phys. Rev.*, 4:345–376, Oct 1914.

[18] F. M. for sustainable development. Sensor Malfunction Fact Sheet. April.

[19] R. T. House. A proposal for an extended form of type checking of expressions. *Comput. J.*, 26(4):366–374, Nov. 1983.

[20] A. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*, ESOP '94, pages 348–362, London, UK, UK, 1994. Springer-Verlag.

[21] A. Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, CEFP'09, pages 268–305, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] M. Kitamura. Detection of Sensor Failures in Nuclear Plants Using Analytic Redundancy. Oak Ridge National Laboratory, Oakridge, Tennessee.

[23] R. Kurth. A note on dimensional analysis. *The American Mathematical Monthly*, 72(9):965–969, 1965.

[24] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 14. ACM, 2014.

[25] P. Stanley-Marbell and D. Marculescu. Sunflower: Full-system, embedded, microarchitecture evaluation. In *Proceedings of the 2Nd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'07, pages 168–182, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] P. Stanley-Marbell and M. Rinard. Lax: Driver interfaces for approximate sensor device access. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[27] P. Stanley-Marbell and M. Rinard. Reducing serial i/o power in error-tolerant applications by efficient lossy encoding. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 62:1–62:6, New York, NY, USA, 2016. ACM.

[28] D. R. Stoutemyer. Dimensional Analysis, Using Computer Symbolic Mathematics. *Journal of Computational Physics*, 24:141–149, June 1977.