CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# KHoon

## A Formal Semantics of Hoon in the $\mathbb{K}$ Framework

Master's thesis in Computer science and engineering

Anton Ahl

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master's thesis 2023

# KHoon

A Formal Semantics of Hoon in the $\mathbb{K}$ Framework

Anton Ahl

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

KHoon: A Formal Semantics of Hoon in the $\mathbb{K}$ Framework
Anton Ahl



Supervisor at Institution: Magnus Myreen, Department of Computer Science and Engineering
Supervisor at Company: Rikard Hjort, Runtime Verification
Examiner: Robin Adams, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

KHoon: A Formal Semantics of Hoon in the $\mathbb{K}$ Framework
Anton Ahl
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Hoon, a programming language with several unusual properties, which is intended for use as a smart contract language, currently lacks a formal semantics, and this poses challenges in terms of verification. This project aims to overcome this issue by providing a formal semantics for Hoon using the $\mathbb{K}$ framework. The resulting specification covers approximately a quarter of the language's features and is accompanied by a test suite consisting of 131 tests to validate the specified features. Furthermore, this paper presents various claims about the Hoon specification that can be proven using the theorem prover in the $\mathbb{K}$ framework. Nonetheless, further work is needed to complete the specification and enable its potential future applications.

Keywords: Formal semantics, Hoon, $\mathbb{K}$ framework, programming languages.

# Acknowledgements

I am deeply grateful to my supervisors, Magnus Myreen and Rikard Hjort, for their invaluable guidance and support throughout this project. Their expertise and feedback have been significant in shaping the outcome of this project. Furthermore, I would like to extend my appreciation to both the Runtime Verification team and the Urbit developers. The comprehensive documentation they have provided about the $\mathbb{K}$ framework and Hoon, respectively, has been immensely beneficial in understanding and effectively leveraging these tools.

Anton Ahl, Gothenburg, 2023-06-08

# Contents

# Glossary

**Arm** An arm is a named Hoon expression.

**Atom** An atom is a non-negative integer.

**Aura** An aura is the type of an atom.

**Battery** A battery is a collection of Hoon expressions.

**Bunt** A bunt value of a type is its default/example value.

**Cell** A cell is an ordered pair of nouns written in square brackets and separated by a single space.

**Core** A core is a cell of a battery and a payload.

**Door** A door is a core whose payload is a cell of a sample and a context.

**Gate** A gate is door with one arm named `$` (buc).

**Leg** A leg is a named piece of data.

**Noun** A noun is an atom or a cell.

**Payload** A payload is the subject of a core.

**Rune** A rune is a digraph consisting of two ASCII special characters, and they serve as the building blocks of Hoon expressions.

**Sample** A sample is the argument taken by a gate or door.

**Subject** The subject is the context against which a specific expression is evaluated.

**Trap** A trap is a core with one arm named `$` (buc).

**Wing** A wing is a resolution path of names separated by the `.` (dot) character.

# List of Figures

# List of Listings

# List of Tables

# 1

# Introduction

A programming language is usually divided into its syntax (the form) and its semantics (the meaning). Formally specifying the syntax of a programming language is common, e.g. by expressing it in Backus–Naur form (BNF). However, it is not as common to formalize the semantics, instead informal explanations combined with examples of programs are often used. Specifying a formal semantics is essential if one is to confidently assert or prove any aspect of the language, such as that a program specification is met by its implementation.

One way to formalize the semantics of a programming language is to use the $\mathbb{K}$ framework. $\mathbb{K}$ is an executable semantic framework [1]. It enables the definition of programming languages, type systems and formal analysis tools. Furthermore, once a formal language definition is specified, $\mathbb{K}$ can derive programming language tools from its semantic specifications. There have been several formal semantics for programming languages that have been specified in $\mathbb{K}$, such as JavaScript [2] and Java [3].

Hoon is a programming language that lacks a formal semantics (for a background on Hoon, refer to Section 2.1). One of Hoon's standout features is its ability to implement a purely functional operating system [4]. Urbit OS is an example of such an operating system, written in Hoon [5], and it is the operating system for the Urbit platform, which is a decentralized network of personal servers [6].

Hoon is also a programming language with several unusual properties, one of which is its subject-oriented nature [4]. In Hoon, each expression is evaluated relative to its *subject*, which is the context within which the expression is evaluated. Intuitively, one can think of this as a perpetual inclusion of both the expression and the context in a closure, with the subject of the expression as the contextual anchor.

There are several reasons why a formal semantics for Hoon is of interest. First, from an academic point of view, every programming language should have formal semantics, and formal analysis tools should be based on these. Second, an unusual aspect of Hoon is that it is subject-oriented, and developing a formal semantics for such a language is interesting in itself. For example, how can it be done effectively? What is challenging about it? In what ways is it easier/harder compared to more conventional languages? Third, Hoon will be used as a smart contract language with Uqbar, a smart contract platform built on top of Urbit [7]. Therefore, verification is important and formal semantics is essential.

## 1.1 Goal

The goal of the project is to specify a formal semantics for Hoon using the $\mathbb{K}$ framework. Additionally, the goal is also to interpret Hoon programs and perform deductive verification using the tools derived from the semantics, once in place. Together, these two goals of the project are encapsulated by the following research question:

> *How can a formal semantics for a subject-oriented language like Hoon be formulated and how suitable is such a semantics for deductive verification?*

Regarding the deductive verification, the goal is to at least start to prove the correctness of some simple programs. For instance, a simple program might be a loop that sums the numbers one through $n$, for some specific positive integer $n$.

Besides the semantics and the deductive verification, another goal of the project is to write a test suite to thoroughly test the semantics. The test suite is intended to be a set of small programs that perform special cases of features or combinations thereof, and one program that runs all tests on the semantics.

## 1.2 Contribution

This project's contributions include:

1. A partial formal executable semantics of Hoon.
2. A test suite consisting of 131 tests that can be executed to verify the features of the specification that have been specified.
3. Claims regarding the Hoon specification that can be proven by the theorem prover in the $\mathbb{K}$ framework.

The semantics is partial as it does not encompass the entirety of the Hoon language. However, to gauge the scope of the semantics without delving too deeply into technical details, it is able to successfully execute Hoon programs such as the one shown in Listing B.1.

The test suite encompasses a set of Hoon programs designed to test the specified features of the language, which is divided into two distinct categories: programs that are expected to terminate successfully and produce a specific result (good tests), and programs that are anticipated to fail in some way, i.e., be rejected (bad tests). Of the total number of tests available, 106 are good tests, while 25 are bad tests.

The proven claims include the correctness of a program that sums the numbers from one to a specific value $n$ (specifically, the program in Listing B.1), as well as claims about the expected behavior of some programs using code from the Urbit kernel and a few other basic claims.

Furthermore, to enhance accessibility and encourage continued development, the

project's code is publicly available in a Git repository on GitHub[1].

## 1.3 Outline

The rest of the paper is structured as follows: Chapter 2 provides background information on the key concepts necessary to comprehend the project. In Chapter 3, our specification of Hoon in $\mathbb{K}$ (KHoon) is clarified, and the test suite we created to rigorously test it is explained. Moving forward to Chapter 4, we present a series of claims regarding our $\mathbb{K}$ specification of Hoon and prove them using the theorem prover in the $\mathbb{K}$ framework. Finally, in Chapter 5, the results of the project are discussed, and a conclusion is provided.

[1] https://github.com/antonahl/khoon

# 2

# Background

This chapter provides background on the topics essential to understanding the project, i.e. the Hoon programming language and the $\mathbb{K}$ framework. While this chapter does not cover everything there is to know about these topics, it (along with later chapters) should be enough to understand the entire project.

## 2.1 The Hoon Programming Language

For those who are not familiar with Hoon, it is important to mention that the subject matter is replete with technical jargon. Therefore, a glossary is included in the frontmatter to aid in understanding these terms. Additionally, to enhance the readability of the report, these terms are italicized and defined upon first use. It is also recommended to study or at least skim through Appendix A to learn how to pronounce Hoon code, as it provides an explanation of the pronunciation.

What is Hoon? Hoon is a strict, statically typed, purely functional programming language. It compilies itself to Nock, which is a Turing-complete function that maps a pair of data and code to its corresponding result. As a comparison, the way Hoon layers over Nock is similar to how C layers over machine code [4], [6], [8].

Hoon programs are built of expressions, a combination of characters that are interpreted and evaluated to produce a value. In Hoon, expressions are constructed using *runes*, symbols consisting of two ASCII special characters (a digraph) that function similarly to keywords in other programming languages. Syntactically, each rune is written in prefix notation and requires a fixed number of "children," or Hoon expression "arguments," which can themselves be runes with their own children. Hoon code chains these together until a value is reached. As a consequence, terminators such as opening and closing parentheses are largely absent from Hoon.

### 2.1.1 Nouns

A *noun* is the most general type of data in Hoon, and it can either be an *atom* or a *cell*. An atom is a non-negative integer (of any size). In addition, Hoon atoms are accompanied by two additional pieces of metadata: an *aura* and an optional constant. The aura specifies the type of an atom, such as whether it represents a floating point or text, and the constant controls whether the type consists of all possible values for that aura or is exactly equal to its own value.

A cell is an ordered pair of nouns written in square brackets and separated by a single space. For example, `[0 1]` is a cell, and so is `[0 [1 [2 [3 [4 5]]]]]`. Since it is common for cells to branch to the right in Hoon, there is a convention that brackets marking the rightmost cells in a running cell can be omitted. This means that the latter of the previous two examples can be written equally as `[0 1 2 3 4 5]`.

### 2.1.2 The Subject

The subject is both a noun and the context against which expressions are evaluated (hence subject-oriented). For each rune in an expression, the subject is exactly what was given to it by the previous rune in the rune chain (a new one is created at each step) and it is an actual piece of data that can be referenced and manipulated.

There are three different ways to access the subject, or indeed any noun:

1. numeric addressing
2. positional addressing
3. *wing* addressing.

Numeric addressing of a noun uses a positive integer to access a specific location of the noun. In Figure 2.1, the numeric addressing of a noun is displayed for the first 15 positive integers, and the pattern continues as expected for larger numeric addresses. For example, in the cell `[1 2 3 4 5]`, the numeric address of the atom `1` is 6. Moreover, it is an error to address a location that is not within the noun, such as address 2 of the atom `1`.

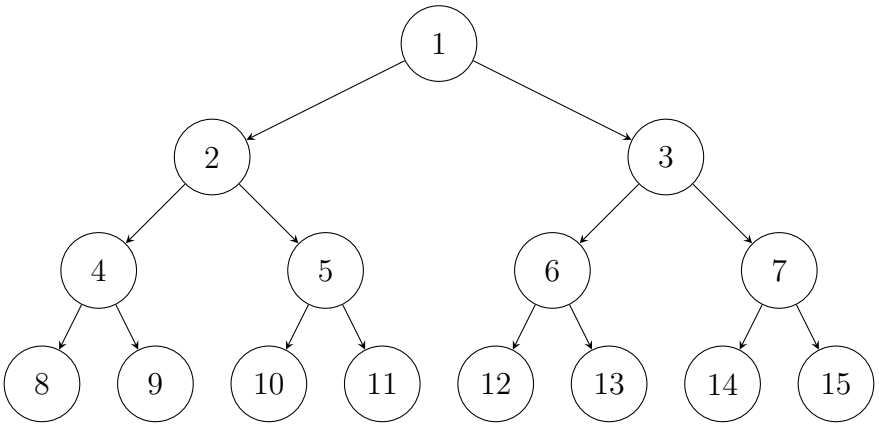



Figure 2.1: Numeric addressing of a noun.

Positional addressing uses directions: left and right. They are written with `-` (left) and `+` (right) alternating with `<` (left) and `>` (right), and `.` (dot) for the root. A demonstration of this is shown in Figure 2.2, which shows how the numeric addresses in Figure 2.1 are equivalently written using positional addressing. That is, in the cell `[0 1 2 3 4 5]`, the positional address of the atom `1` is `+<`.

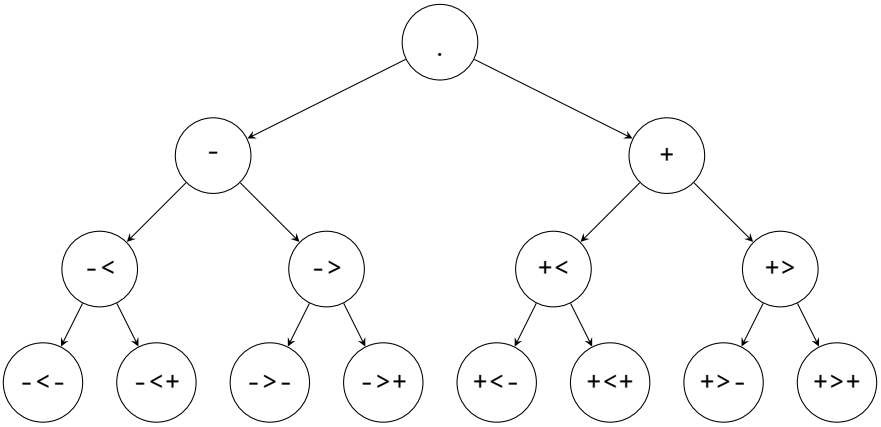



Figure 2.2: Positional addressing of a noun.

Wing addressing uses wings to access a resource in a noun (using depth-first search, starting with the left branch). A wing is a resolution path of names separated by the `.` (dot) character. For example, `a.b.c` is a wing and can be read as `a` in `b` in `c`. If a wing resolves to an *arm* (a named computation), the computation is run and the result is returned, otherwise, if a wing resolves to a *leg* (named data), the data is returned.

### 2.1.3 Cores

A *core* is a cell, where the left noun is called the *battery* and the right noun is called the *payload*. The battery of a core is a collection of arms, which contain the computations that can be performed, while the payload is the data on which those computations rely, i.e. the subject of the core.

The arms of a battery are Hoon expressions encoded as nouns, or rather compiled into Nock formulas. Nock formulas are nouns used as functions that take a noun and its subject, and return a noun. Typically, one can simply think of arms as a way to run some Hoon code.

A core that has a battery consisting of one arm named `$` (buc) and a payload that is a cell of a *sample* and a context, is called a *gate* (see Figure 2.3). Gates are commonly referred to as "Hoon functions" because they share many characteristics with functions from other programming languages. The sample of a gate is the argument that is passed to it, and the context includes all other necessary data for correctly computing the `$` (buc) arm of the gate.

```
  gate
 /    \
$      .
      / \
sample   context
```

Figure 2.3: The structure of a gate.

Another type of core is the *door*. A door is a core with a sample. This means that a

gate is simply a special case of a door - a door with only one arm, called `$` (buc). To clarify, a door can have a variable number of arms (with a minimum of one), while a gate is limited to only one arm named `$` (buc). Figure 2.4 illustrates the structure of a door.

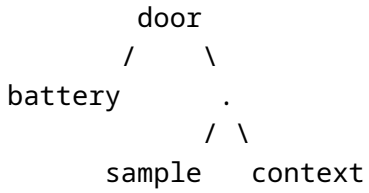



Figure 2.4: The structure of a door.

### 2.1.4 Rune Families

Runes are categorized into families, and Table 2.1 provides an overview of the families present, the runes found in each family, and a brief description of the family. The first of the two ASCII characters in a rune indicates the family to which the rune belongs (except the terminator family). For example, the `=/` (tisfas) rune is in the `=` (tis) family, and the `|-` (barhep) rune is in the `|` (bar) family. Additionally, runes in a family usually have related meanings. For instance, runes in the `|` (bar) family are used to produce cores, and runes in the `=` (tis) family are used to modify the subject.

Table 2.1: A table of the rune families in Hoon.

| Family | | Runes | Description |
|---|---|---|---|
| `+` lus | (Arms) | `+\|` `+$` `++` `+*` | define arms in a core |
| `\|` bar | (Cores) | `\|$` `\|_` `\|:` `\|%` `\|.` `\|^` `\|-` `\|~` `\|*` `\|=` `\|@` `\|?` | produce cores |
| `$` buc | (Structures) | `$\|` `$_` `$%` `$:` `$<` `$>` `$-` `$^` `$&` `$~` `$@` `$=` `$?` | define custom types |
| `%` cen | (Calls) | `%_` `%:` `%.` `%-` `%^` `%+` `%~` `%*` `%=` | make "function calls" |
| `:` col | (Cells) | `:_` `:-` `:^` `:+` `:~` `:*` `::` | produce cells |
| `.` dot | (Nock) | `.^` `.+` `.*` `.=` `.?` | perform Nock operations |
| `/` fas | (Imports) | `/$` `/%` `/-` `/+` `/=` `/*` `/~` `/?` | import files |
| `^` ket | (Casts) | `^\|` `^:` `^.` `^-` `^+` `^&` `^~` `^*` `^=` `^?` | adjust types |
| `;` mic | (Make) | `;:` `;/` `;<` `;+` `;;` `;~` `:*` `;=` | various macros |
| `~` sig | (Hints) | `~\|` `~$` `~_` `~%` `~/` `~<` `~>` `~+` `~&` `~=` `~?` `~!` | information to the interpreter |
| `=` tis | (Subject) | `=\|` `=:` `=,` `=.` `=/` `=<` `=>` `=-` `=^` `=+` `=;` `=~` `=~` `=?` | modify the subject |

Table 2.1: (continued)

| Family | Runes | Description |
|---|---|---|
| `?` wut (Conditionals) | `?\|` `?:` `?.` `?<` `?>` `?-` `?^` `?+` `?&` `?@` `?~` `?=` `?!` | branching on conditionals |
| `!` zap (Wild) | `!:` `!,` `!.` `!<` `!>` `!;` `!@` `!=` `!?` `!!` | wildcard category |
| `--`, `==` (Terminators) | `--` `==` | terminate expressions |

### 2.1.5 Tall and Wide Forms

There are two ways to write rune expressions in Hoon: tall form and wide form, and most runes can be written in either form. In tall form, rune expressions are separated by gaps, which are two or more spaces or a line break. In contrast, in wide form, the rune expression is directly followed by parentheses, and the subexpressions inside the parentheses are separated by aces, which are single spaces. For example, to construct a cell of `0` and `1` with the `:-` (colhep) rune, which constructs a cell from its two children, in tall and wide form respectively, one could write:

```
:-  0
1
```

or

```
:-(0 1)
```

Note that tall form expressions can contain subexpressions in wide form, but the reverse is not true.

### 2.1.6 An Example Program

To clarify the background on Hoon so far, a simple Hoon program is explained line by line, namely the one in Listing 2.1. It is a program that sums the numbers one to five. In lines 1 and 2, the `=/` (tisfas) rune names the nouns `1` and `0` to `counter` and `sum`, respectively, and combines them with the subject (similar to how assignment works in other programming languages). Line 3 produces a *trap* (a core with one arm named `$` (buc)) and evaluates it. In this example, the arm contains the code on lines 4-9. Line 4 checks whether the value of `counter` is greater than five by calling the `gth` gate (which must be in the subject) with the sample `[counter 5]`. An irregular form of the `%-` (cenhep) rune is used here; `(gth counter 5)` could correspondingly be written as `%-(gth [counter 5])`. If the value of `counter` is greater than five, `sum` is evaluated, i.e., its value is looked up in the subject. Otherwise, lines 6-9 are evaluated. The `%=` (centis) rune used here looks up the `$` (buc) arm and evaluates it, but with the following changes made:

1. The new value of `counter` is the old `counter` value plus one.
2. The new value of `sum` is the sum of the old `sum` and old `counter` values.

Intuitively, one can think of the program as first pinning the `counter` and `sum` nouns to the subject. Then, the runes `|-` (barhep) and `%=` (centis) work together to create a looping structure: if the conditional test on line 4 evaluates to false, the program

makes the changes on lines 7-8 and returns to the test on line 4 again. Once the test evaluates to true, the sum is returned.

```
1 =/  counter  1
2 =/  sum  0
3 |-
4 ?:  (gth counter 5)
5   sum
6 %=  $
7   counter  (add counter 1)
8   sum      (add sum counter)
9 ==
```

Listing 2.1: A Hoon program that computes and returns the sum 1+2+3+4+5. (The gates `gth` and `add` are assumed to already exist in the subject before evaluation.)

## 2.2 The $\mathbb{K}$ framework

$\mathbb{K}$ is an executable semantic framework that enables the specification of programming languages, type systems, and formal analysis tools [1], [9]. A $\mathbb{K}$ specification is a collection of sentences, which are organized into modules and files. These sentences can be syntax, configuration, and rule declarations. In short, syntax declarations encode the syntax, configuration declarations encode the system state, and rule declarations encode the system behavior. A more thorough explanation of this is given in the three subsequent subsections.

Modules group sentences and are stored in files. A file can contain one or more modules and can also import modules from other files. Note however, the resulting dependency graph obtained from this must form a directed acyclic graph so as not to create cyclic dependencies.

Once a $\mathbb{K}$ specification is formulated, the $\mathbb{K}$ framework provides several tools for the specification. These tools are:

**kompile** Generates a parser from the given $\mathbb{K}$ specification.

**kast** Parses an expression or a file.

**kparse** Parses and unparses a file.

**krun** Interprets a file or a string.

**kprove** Verifies claims in a file.

$\mathbb{K}$ also has built-in functionality, a set of files containing definitions to facilitate the writing of $\mathbb{K}$ specifications. For example, there is an implementation for booleans, integers, and strings, among others, along with their most common operations [10].

### 2.2.1 Syntax Declarations

Syntax declarations encode the syntax and are constructed with the `syntax` keyword and have a BNF style, optionally with attributes. Attributes are written in brackets separated by commas after the production it is associated with and indicate something about the production.

An example of three syntax declarations that demonstrate some of the functionality of $\mathbb{K}$ is shown in Listing 2.2. It defines a syntax for adding integers and summing lists of integers. Furthermore, it is contained in a module called `ADDER-SYNTAX` and imports the module `INT-SYNTAX`, which is a built-in module that defines a syntax for integers.

```
1  module ADDER-SYNTAX
2      imports INT-SYNTAX
3
4      syntax Exp ::= Int
5                   | Exp "+" Exp [strict, left]
6
7      syntax Ints ::= List{Int, ","}
8
9      syntax Int ::= sum(Ints) [function]
10 endmodule
```

Listing 2.2: An example syntax for a $\mathbb{K}$ specification for adding integers and summing lists of integers.

The first syntax declaration (lines 4-5) declares two production rules for the nonterminal `Exp`; it is either an `Int` or two `Exp`s with a plus symbol in between. The second production has two attributes: `strict` and `left`. These mean that the production will use a strict evaluation strategy for the subterms and be left associative with itself, respectively.

The second syntax declaration on line 7 declares `Ints` as an arbitrarily long comma-separated list of integers. The list construct used here is roughly equivalent to `syntax Ints ::= Int "," Ints | ".Ints"`, where `.Ints` is the empty `Ints` list. Note that it is possible to define syntactic lists with regular productions, the list construct just makes it more convenient.

The third syntax declaration declares that an `Int` can also be written as `sum(...)` where there is a list of integers between the parentheses (instead of the dots), for example `sum(1, 2, 3)`. The associated attribute (`function`) means that the expression will be simplified immediately when it appears in the configuration, in accordance with the matching rules. In this example, these rules are specified on lines 13-14 in Listing 2.4.

### 2.2.2 Configuration Declarations

Configuration declarations encode the system's state. They are constructed using the `configuration` keyword and a hierarchy of cells, which means that the cells can be nested or placed next to each other. It is also possible to assign attributes to cells, such as setting the exit code for invocations of `krun` or allowing multiple copies of the same cell. To clarify, a cell in $\mathbb{K}$ is not the same as a cell in Hoon. The context determines which one the term refers to. For example, this section discusses cells in $\mathbb{K}$, while Section 2.1.1 discusses cells in Hoon.

The example shown in Listing 2.3 shows the first part of a module called `ADDER`, which imports the preceding module `ADDER-SYNTAX`, as well as the built-in module `INT`. The `INT` module is a module that imports the `INT-SYNTAX` module and also includes common integer arithmetic operations. The `ADDER` module also includes a single configuration declaration that declares three cells: `<k>`, `<counter>`, and `<T>`. The `<k>` cell is initialized to the value of the `$PGM` configuration variable (which serves as input to `krun`) at the start of rewriting, and casts it to the `Exp` sort. The `<counter>` cell is initialized to the integer 0, while the `<T>` cell simply contains the `<k>` and `<counter>` cells.

```
1 module ADDER
2     imports ADDER-SYNTAX
3     imports INT
4
5     configuration
6         <T>
7             <k> $PGM:Exp </k>
8             <counter> 0:Int </counter>
9         </T>
```

Listing 2.3: The configuration of the previous example (Listing 2.2).

The `<k>` cell is used to contain a series of computations, ending with a period and sequenced with the operator: `~>`. Since it is initialized to the value of `$PGM`, the `<k>` cell will initially hold the given input (the program code). Furthermore, if one does not explicitly specify a configuration declaration in the main module, the default configuration will be used, and it declares the following:

```
configuration <k> $PGM:K </k>
```

Thus, in Listing 2.3 the configuration have only been extended with two more cells (nested as shown in the listing) and the sort of the `$PGM` configuration variable limited to an `Exp` (each sort is a subsort of `K`), compared to the default configuration.

### 2.2.3 Rule Declarations

Rule declarations encode the behavior of the system. They are constructed with the `rule` keyword followed by a left-hand side (LHS), the rewrite operator (`=>`), the right-hand side (RHS), and optionally with a requires clause, ensure clause and/or

a comma-separated list of attributes in brackets. If one rule is matched, the LHS is rewritten to the RHS. A rule is matched if the LHS pattern is matched and the requires clause is met. The ensures clause is interpreted as a post-condition that should hold and attributes indicate something about the rule. In the case that multiple rules are matched, it is non-deterministic which rule applies. To enforce a specific rule in such a scenario, priority attributes can be used to prioritize the rules.

Listing 2.4 contains a syntax declaration and three rule declarations and is a continuation of the module in Listing 2.3. The syntax declaration is necessary because of the `strict` attribute used in the `ADDER-SYNTAX` module. The `strict` attribute generates *heating* and *cooling* rules for the associated expression, and to make this work, one must define what a result is by extending the `KResult` sort. This is necessary to know when to stop the heating process. (In Listing 2.4, a result is defined as an `Int`.) Heating and cooling rules split the syntax into a redex (the head of the computation) and an evaluation context (the tail of the computation) and plug the syntax into the context, respectively. For example `1 + 2` is "heated" to `1 ~> □ + 2`, while `1 ~> □ + 2` is "cooled" to `1 + 2`. Here, the □ represents a placeholder for the syntax that has been heated.

```
8     syntax KResult ::= Int
9
10    rule <k> I1 + I2 => I1 +Int I2 ... </k>
11         <counter> I => I +Int 1 </counter>
12
13    rule sum(I:Int) => I
14    rule sum(I1:Int, I2:Int, Is:Ints) => sum(I1 +Int I2, Is)
15 endmodule
```

Listing 2.4: The continuation of the `ADDER` module in Listing 2.3, which contains the rule declarations.

The first rule (line 10) is matched if the head of the `<k>` cell matches the pattern `I1 + I2` and if the `<counter>` contains an `Int`. Due to the `strict` attribute associated with this syntax, `I1` and `I2` will evaluate to integers (`Int`). The rule will then apply, rewriting `I1 + I2` to `I1 +Int I2`, which sums integers (`+Int` is a built-in function that computes the sum of two integers) and incrementing the integer in the `<counter>` cell by one. The three dots at the end of the `<k>` cell part of the rule mean that the rule does not "care" about the rest of the `K` sequence and leaves it unchanged.

The other two rules compute the sum of the given list of integers. The rule on line 12 rewrites the sum of a singleton integer list to the integer it contains, and the rule on line 13 rewrites the sum of an integer list of length at least two to the sum of the same integer list but with the first two integers summed.

### 2.2.4 Claims

Claims serve as a notation to specify the proper operation of specific $\mathbb{K}$ programs. They are written in a similar way to the rule declarations, but with the `claim` keyword instead, and are checked by the theorem prover (`kprove`) provided by the $\mathbb{K}$ framework, whose infrastructure is based on reachability and matching logic [11].

Listing 2.5 shows a file that imports the `ADDER` module from Listing 2.3 and Listing 2.4 and contains a simple claim. The claim is that one plus two plus the sum of three through five rewrites to 15 and that the counter rewrites to two, assuming it is zero to begin with.

```
1 requires "adder.k"
2
3 module ADDER-SPEC
4     imports ADDER
5
6     claim <k> 1 + 2 + sum(3, 4, 5) => 15 </k>
7           <counter> 0 => 2 </counter>
8 endmodule
```

Listing 2.5: A simple claim about the `ADDER` specification.

### 2.2.5 Using the Derived Tools

A $\mathbb{K}$ specification is compiled with `kompile`. It has options to generate an ahead of time parser instead of just in time (the default), enabling non-deterministic runs and choosing the backend among other things. Once a $\mathbb{K}$ specification has been compiled, the remaining tools can be used. For example, running `kast` and `kparse` on a file containing only the integer 1 with a compiled `ADDER` specification (listings 2.3 and 2.4) is shown in listing 2.6 and 2.7, respectively.

```
$ kast one.adder
#token("1","Int")
```

Listing 2.6: Running `kast` on a file containing only the integer 1 with a compiled `ADDER` specification.

```
$ kparse one.adder
inj{SortInt{}, SortExp{}}(\dv{SortInt{}}("1"))%
```

Listing 2.7: Running `kparse` on a file containing only the integer 1 with a compiled `ADDER` specification.

To interpret a string or a file, `krun` is used. For example, in Listing 2.8, the string '`1 + 2 + sum(3, 4, 5)`' is interpreted, resulting in the value 15. It is also possible to step through the rewriting process using the depth option (starting at 0). This is demonstrated in Listing 2.9 by executing only a single top-level rule to the same

input string as Listing 2.8 (by giving the `--depth 1` option). In the output, one can see that `sum(3, 4, 5)` already has been evaluated (due to its `function` attribute) and that the top-level rule that has been applied is a heating rule, turning `1 + 2 + 12` into `1 + 2 ~> □ + 12`, which also shows the left associativity of the expression.

```
$ krun -cPGM='1 + 2 + sum(3, 4, 5)'
<T>
  <k>
    15 ~> .
  </k>
  <counter>
    2
  </counter>
</T>
```

Listing 2.8: Interpreting the input string '`1 + 2 + sum(3, 4, 5)`' with a compiled `ADDER` specification.

```
$ krun -cPGM='1 + 2 + sum(3, 4, 5)' --depth 1
<T>
  <k>
    1 + 2 ~> #freezer_+__ADDER-SYNTAX_Exp_Exp_Exp0_ ( 12 ~> . ) ~> .
  </k>
  <counter>
    0
  </counter>
</T>
```

Listing 2.9: Interpreting the same input string as in Listing 2.8 with a compiled `ADDER` specification but only performing one rewrite step.

The theorem prover `kprove` is used to verify claims about a $\mathbb{K}$ specification. For example, one can verify the earlier claim (Listing 2.5), as shown in Listing 2.10. The resulting output `#Top` from running `kprove` means that the claim has been proven to be correct. However, note that the theorem prover may not be able to prove a claim directly on its own if it is more sophisticated. In this case, certain simplification rules must be specified to substantiate the claims, and it is the developer's responsibility to ensure that the provided simplification rules are sound, as this is not controlled by the frontend or backend.

```
$ kprove adder-spec.k
#Top
```

Listing 2.10: Proving the claim in Listing 2.5.

# 3

# Formal Specification

In this chapter, we will explain our specification of Hoon in $\mathbb{K}$ (KHoon) and the test suite we developed to test it. We will start by providing an overview of the various components involved in writing the specification, followed by more thorough explanations of each individual part. However, the entire $\mathbb{K}$ specification is not explained in minute detail (see the specification itself for this), instead the primary focus is on the key aspects of the specification, providing sufficient detail to understand its most important parts.

## 3.1 Overview

There are three main parts involved in the specification: the preparser, KHoon, and the pretty-printer. Figure 3.1 illustrates how these three parts are connected. The preparser takes a file containing Hoon code as input and modifies it into a format that is accepted by the KHoon syntax. The reason for this is to enable the $\mathbb{K}$ specification to distinguish between aces (a single space) and gaps (two or more spaces or a line break), and to replace characters that have special meaning in $\mathbb{K}$. KHoon is the $\mathbb{K}$ specification of Hoon, which contains both the syntax and the semantics. Finally, the pretty-printer trims the resulting output from executing the semantics and reverses the changes made by the preparser, in order to return an actual Hoon value.

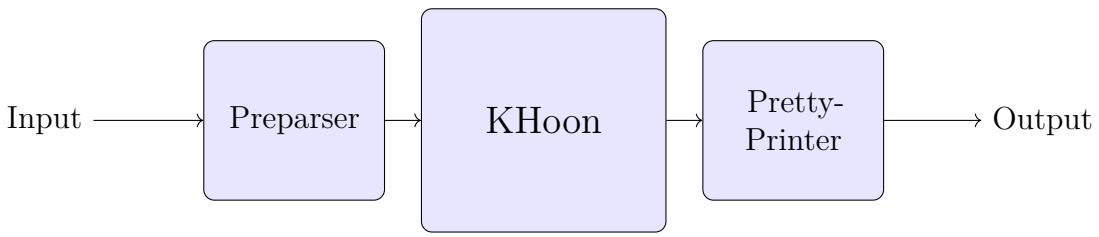



Figure 3.1: An abstract overview of the various components involved in writing a specification and how they are connected. It covers the process from input to output, where the input is the Hoon program to be executed, and the output is the resulting value (a noun) obtained from executing the program.

## 3.2 Preparser

The preparser is a program that replaces certain symbols in the given Hoon code with something else (by utilizing regular expressions). Specifically, these symbols are aces, gaps, and dots. Each ace is replaced by the string `"\ace"`, each gap by the string `"\gap"`, and each dot by the string `"\dot"`, but only if the meaning of the dot is to get the root of a noun, i.e. a positional address. For instance, in Hoon, the wing "`...`" means the root of current subject (the first dot) in (the second dot) the root of current subject (the third dot), and would instead be written `\dot.\dot` after preparsing.

## 3.3 KHoon

As previously mentioned, KHoon is our specification of Hoon in $\mathbb{K}$, encompassing both its syntax and semantics, and the following subsections delve into how the key aspects of the specification are specified. These are non-rune expressions, cells, the subject, types, cores, and function calls.

### 3.3.1 Non-Rune Expressions

Non-rune expressions are expressions without runes, and these include atoms and wings. There are also other non-rune expressions besides these but they are not currently part of the specification and are therefore not discussed here. As mentioned earlier, an atom is a non-negative integer, and it has an aura attached to allow the representation of things other than non-negative integers. An aura starts with `@` (pat) followed by zero or more letters that determine what kind of representation the atom should have. Examples of auras are `@tas`, which is ASCII text prefixed with `%` (cen) and called terms, and `@ud` and `@ui`, which are two different ways of writing unsigned integer decimal values. In fact, these are exactly the different types of atoms that the specification currently supports.

To specify the different types of atoms, each one has its own sort. This is accomplished by using a syntax declaration along with the `token` attribute, which allows the syntax to be specified with a regular expression. For example, consider how `@ud` atoms (non-negative integers with dots as thousands separators, e.g. `0` and `12.345`) are specified (refer to Listing 3.1). The sort of these atoms, `UD`, is specified by the regular expression on line 3, which is named `UD`.

```
1 syntax UD ::= r"{UD}" [token]
2 syntax lexical UD =
3     r"(0{1})|([1-9]{1}[0-9]{0,2})(\\.(\\\\gap)?[0-9]{3})*"
```

Listing 3.1: The syntax for `@ud` atoms.

How are these different sorts connected to the actual non-negative integer value of the atom? Some runes can compare atoms with different auras by comparing their

non-negative integers, so to specify these runes there must be a way to also acquire the non-negative integer of an atom, regardless of its aura. This is done by using the built-in functionality of the $\mathbb{K}$ framework that allows conversion between tokens and strings and between strings and integers in a given radix. For instance, the non-negative integer for `@tas` atom `%ab` is 25185, or 0x6261 in hexadecimal (the ASCII values of the letters reversed), and to obtain it the steps in Figure 3.2 are taken.

| **Token:** | `%ab` | ⟶ | `"ab"` | ⟶ | `"6261"` | ⟶ | `25185` |
|---|---|---|---|---|---|---|---|
| **Sort:** | `TAS` | | `String` | | `String` | | `Int` |

Figure 3.2: The conversion steps from a `@tas` atom to its non-negative integer.

Wings are another type of non-rune expression. A wing is one of the addressing types discussed in Section 2.1.2, and is a resolution path of names separated by dots, such as `a.b.c`. However, the term 'wing' is also used for mixed addressing types, such as `+3.-.c` (a numeric address followed by a positional address followed by a name), because these different addressing types can be combined.

Listing 3.2 shows the syntax for wings. It can either be a numeric address (`NumAddr`), a positional address (`Lark`), a name (`Name`), a name prepended with one or more `^` (kets) (`KetName`), or two wings separated by a dot which is right-associative. (Prepending a name with $x$ number of `^` (kets) means that the first $x$ matches of the name that are looked up are ignored. For instance, looking up `^a` in the noun `[[[a=1 a=2] a=3] a=4]` would return 2, not 1.) The syntax for these different sorts is specified in the same way that the different atoms were specified, i.e. using regular expressions.

```
1 syntax Wing ::= NumAddr
2               | Lark
3               | Name
4               | KetName
5               | Wing "." Wing [right]
```

Listing 3.2: The syntax for wings.

Wings of the last kind in Listing 3.2 (line 5) are processed from right to left, and names are resolved using DFS (starting with the left branch), and numeric and positional addressing works as shown in Figures 2.1 and 2.2 of section 2.1.2. For instance, consider the wing `+3.-.c`. In this case, the name `c` would be resolved first, and then the left part of the resulting noun (cell) would be taken (because of the positional address `-`). Then, the noun located at the third address of that resulting noun would be returned. Additionally, there are two ways to use a wing: by itself or with a specific noun. A wing by itself is resolved against the subject while a wing with a specific noun is resolved against the specific noun. (The syntax for the latter is shown in Listing 3.3.)

```
1 syntax Lookup ::= Wing ":" Hoon [seqstrict(2)]
2                 | Noun ":" Hoon
```

Listing 3.3: The syntax for resolving a wing against a specific noun. The `Hoon` sort covers any Hoon code, the `Noun` sort is for nouns, and the `seqstrict` attribute here generates rules to strictly evaluate the second argument (`Hoon`) to a noun.

The rules that specify this behavior are shown in Listing 3.4, which cover the two ways of using a wing: by itself (lines 2-7) and with a noun (lines 10-15). The behavior of a wing that is by itself is specified by two rules, both of which have the same meaning. The only difference is that the first rule matches wings within a tall form expression, while the second matches wings within a wide form expression. This pattern of writing the same rule twice, once for tall form expressions and once for wide form expressions, is common throughout the specification. The rules rewrite a wing to the lookup function, with the subject as the noun supplied to the function.

```
1  // Lookup wings in the subject
2  rule <k> (P:Wing):TallExp
3       =>  lookup(toWingStack(P, .WingStack), S, .MetaNounStack, 0)
4       ... </k>
5       <subject> S->_ </subject> [priority(50)]
6  rule <k> (P:Wing):WideExp
7       =>  lookup(toWingStack(P, .WingStack), S, .MetaNounStack, 0)
8       ... </k>
9       <subject> S->_ </subject> [priority(51)]
10
11 // Wing the specified noun
12 rule <k> (P:Wing : Q):TallExp
13      =>  lookup(toWingStack(P, .WingStack), Q, .MetaNounStack, 0)
14      ... </k> [priority(50)]
15 rule <k> (P:Wing : Q):WideExp
16      =>  lookup(toWingStack(P, .WingStack), Q, .MetaNounStack, 0)
17      ... </k> [priority(51)]
18 rule <k> (P:Noun : _):TallExp => P ... </k> [priority(50)]
19 rule <k> (P:Noun : _):WideExp => P ... </k> [priority(51)]
```

Listing 3.4: The rules of wings.

The lookup function finds the noun in the given noun that the wing refers to. Before calling it, the given wing is converted into a stack of wings. For example, the wing `a.b.c` would be converted into the wing stack `c->b->a`. To process the entire wing stack, the lookup function recursively looks up the tail of the wing stack in the resulting noun obtained from looking up the head of the stack. For instance, when looking up `c->b->a`, the first step would be to look up `b->a` in the noun resulting from looking up `c`. Each of the four cases is then handled accordingly:

1. If the wing is a numeric address, the address is converted to the corresponding positional address. This address is then used to lookup the wing.

2. If the wing is a positional address, the directions are simply followed to find the referenced noun.
3. If the wing is a name, DFS (starting with the left branch) is used to find the name in the noun. For instance, in the case shown in Figure 3.3, it would push noun `R` to the noun stack (the third argument to the function and initially empty) for later processing and start searching for the noun in noun `L`.
4. If the wing is a name that is prefixed with one or more `^` (kets), DFS is employed in the same way as with regular names. However, the fourth argument to the function, which indicates the number of matches to ignore, is adjusted to the number of `^` (kets) the name is prefixed with.

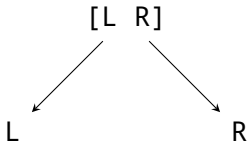



Figure 3.3: The cell `[L R]` illustrated as a tree, where L and R are metavariables for arbitrary nouns.

The behavior of a wing with a noun (the other way of using a wing) is similar to that of a wing by itself, only here the noun supplied to the lookup function is the noun that is associated with the wing. Also, if the wing is replaced with a noun, it is simply rewritten to that noun.

### 3.3.2 Cells

A cell is an ordered pair of nouns written within square brackets and separated by a single space, and their specified syntax is shown in Listing 3.5. The syntax covers the two methods of writing a cell: including all the brackets and omitting the brackets that mark the rightmost cell in a cell.

```
1 syntax Cell ::= "[" Hoon "\\ace" Hoon "]" [seqstrict, prefer]
2               | "[" Hoon "\\ace" Hoon "\\ace" CellImplicit "]"
3
4 syntax CellImplicit ::= List{Hoon, "\\ace"}
```

Listing 3.5: The syntax for cells. Notice that a cell is a noun pair but the syntax here allows any Hoon code in a cell. This is allowed and intentional because Hoon code written in a cell will strictly evaluate to a noun.

To eliminate the need to handle two distinct syntactic cases for cells with identical semantics, any cell that omits brackets is converted to its corresponding version that includes all brackets. The rules specifying this behavior are shown in Listing 3.6. They specify that a cell with omitted brackets (line 3-4) is rewritten to the explicit brackets function, which adds all the missing brackets. This rule has a priority level of 49, which is higher than the default priority of 50, and therefore takes precedence

over any other rule without a priority attribute or one with a lower priority.

```
1 syntax Cell ::= explicitBrackets(Cell) [function]
2
3 rule [P \ace Q \ace R]:Cell
4      => explicitBrackets([P \ace Q \ace R]) [priority(49)]
5 rule explicitBrackets([P \ace Q \ace R \ace REST]:Cell)
6      => [P \ace explicitBrackets([Q \ace R \ace REST])]
7 rule explicitBrackets([P \ace Q \ace .CellImplicit]:Cell)
8      => [P \ace Q]
```

Listing 3.6: The rules for rewriting a cell with omitted brackets into its corresponding version that includes all brackets.

### 3.3.3 The Subject

The subject is specified as a stack of *metanouns* (nouns with metadata attached) and resides in a configuration cell (refer to Listing 3.7). The reason for using a stack of nouns instead of just one noun to manage the subject is to enable restoration of the subject after evaluating nested expressions. For instance, the Hoon expression in Listing 3.8 cannot be evaluated without saving the state of the subject at the first `=>` (tisgar). Figure 3.4 illustrates how the expression is nested. This is because the second `=>` (tisgar) on line 2 sets the subject to the noun `2`. However, in line 3, the current subject should again be the noun `1` because of the first `=>` (tisgar) on line 1.

```
1 configuration
2     <k> $PGM:Hoon </k>
3     <subject> (nullMetaNoun()->.Subject):Subject </subject>
4
5 syntax Subject ::= List{MetaNoun, "->"}
```

Listing 3.7: The configuration declaration of our specification. It consists of two cells: the `<k>` cell and the `<subject>` cell. The `<k>` cell is initialized to the Hoon code that is input to the generated interpreter (`krun`), and the `<subject>` cell is initialized to a stack of one noun that is null.

```
1 =>  1
2 :-  =>(2 .)
3 .
```

Listing 3.8: A Hoon expression that evaluates to `[2 1]`. To clarify why, `=>` (tisgar) takes two children and evaluates the second with the first as subject, `:-` (colhep) produces a cell from its two children, and the `.` (dot) gets the current subject.

To manipulate the subject, the runes from the `=` (tis) family are used. Most runes in the `=` (tis) family expand to `=>` (tisgar) combined with other runes, and currently,

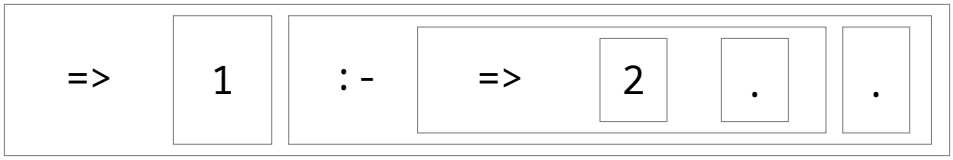



Figure 3.4: An illustration of how the expression in Listing 3.8 is nested.

all of the specified `=` (tis) runes behave this way. For example, `=+(p q)` (tislus) expands to `=>([p .] q)` and `=|(p q)` (tisbar) expands to `=+(^*(p) q)`.

The syntax and rules of `=>` (tisgar) are shown in Listings 3.9 and 3.10, respectively. Like most runes, `=>` (tisgar) can be written in both tall and wide form, and the semantics are the same regardless of the form used. The rules specify that `=>` (tisgar) rewrites to a `push` of its first child, followed by the evaluation of the second child and a `pop`.

```
1 syntax TallForm ::= "=>" "\\gap" Hoon "\\gap" Hoon
2
3 syntax WideForm ::= "=>" "(" WideExp "\\ace" WideExp ")"
```

Listing 3.9: The syntax of `=>` (tisgar).

```
1 rule <k> => \gap P \gap Q => push(P) ~> Q ~> pop() ... </k>
2 rule <k> => (P \ace Q) => push(P) ~> Q ~> pop() ... </k>
```

Listing 3.10: The rules of `=>` (tisgar).

A `push` (specified in Listing 3.11) pushes the given argument to the subject stack and then proceeds to process the tail of the `<k>` cell. However, before the argument is pushed, it is first evaluated to a noun, and it is the context declaration on line 4 that generates the heating and cooling rules that takes care of this. Correspondingly, a `pop` (specified in Listing 3.12) pops the top of the subject stack, and it does so if the head of the `<k>` cell has evaluated to a noun and is followed by a `pop`.

```
1 syntax Hoon ::= push(Hoon)
2 rule <k> push(P) ~> K => K </k>
3      <subject> S => P->S </subject>
4 context push(HOLE:Hoon)
```

Listing 3.11: The specification of `push`.

```
1 syntax Hoon ::= pop()
2 rule <k> P:MetaNoun ~> pop() ~> K => P ~> K </k>
3      <subject> _->S => S </subject>
```

Listing 3.12: The specification of `pop`.

### 3.3.4 Types

Part of the metadata in a metanoun is type information (the specification of metanouns is shown in Listing 3.13). Types in Hoon can take various forms, including auras, terms, and cells of types. For example, the `@ud` aura is a type for unsigned integers and `[%foo @]` is a type for cells of the term `%foo` and any atom. Regarding terms, they are atoms whose type is an exact value equal to itself, i.e. the type of the term `%foo` is `%foo`. To accommodate the different types, the syntax specification `Spec` in Listing 3.14 is used.

```
1 syntax MetaNoun ::= "(" Meta "," Noun ")"
2
3 syntax Meta ::= "{" Spec ";" ArmNames "}"
```

Listing 3.13: The syntax for metanouns.

```
1  syntax Spec ::= "[" Spec "\\ace" Spec "]"
2                | "[" Spec "\\ace" Spec "\\ace" SpecCellImplicit "]"
3                | TAS
4                | "*"
5                | "@"
6                | "^"
7                | "?"
8                | "~"
9                | Aura
10               | Name "="
11 syntax SpecCellImplicit ::= List{Spec, "\\ace"}
```

Listing 3.14: The syntax for types (`Spec`).

The reason for keeping track of types with metanouns is to enable the specification of ket runes (runes used to adjust types). Some ket runes expand to other ket runes, possibly in combination with runes from other rune families, and out of the currently specified ket runes, the three that do not expand to other runes are `^+` (ketlus), `^*` (kettar), and `^=` (kettis). Their syntax is specified in Listing 3.15.

```
1  syntax TallForm ::= "^+" "\\gap" Hoon "\\gap" Hoon [seqstrict]
2                    | "^*" "\\gap" Spec
3                    | "^=" "\\gap" Skin "\\gap" Hoon [seqstrict(2)]
4
5  syntax WideForm ::= "^+" "(" WideExp "\\ace" WideExp ")" [seqstrict]
6                    | "^*" "(" Spec ")"
7                    | "^=" "(" Skin "\\ace" WideExp ")" [seqstrict(2)]
```

Listing 3.15: The syntax for `^+` (ketlus), `^*` (kettar), and `^=` (kettis).

The `^+` (ketlus) rune (specified in Listing 3.16) is typecast by inferred type. It takes two children $p$ and $q$, and produces the value of $q$ with the type of $p$, if the type of $q$ nests within the type of $p$. In the specification, this is done by utilizing the type

information contained in the metadata, as well as the `toType` and `nests` functions. The former formats a noun according to the given type (for instance, the term `%ab` formatted with the `@` aura is `25.185`), while the latter checks whether the second type nests within the first.

The type information stored in a metanoun is inferred by observing the sort of a noun token at the time the metadata is constructed. For instance, the atom `12.234` has the `UD` sort (as shown in Listing 3.1), so when the noun is converted to a metanoun and the metadata is constructed, the type information will be `@ud`. However, this initial type information can be overridden by using some (ket) rune on the metanoun.

```
1 rule <k> ^+ \gap ({T1; _}, _) \gap ({T2; _}, Q)
2      =>  ({T1; .ArmNames}, toType(T1, Q))
3      ... </k> requires nests(T1, T2)
4 rule <k> ^+(({T1; _}, _) \ace ({T2; _}, Q))
5      =>  ({T1; .ArmNames}, toType(T1, Q))
6      ... </k> requires nests(T1, T2)
```

Listing 3.16: The rules of `^+` (ketlus).

The `^*` (kettar) rune (specified in Listing 3.17) produces a *bunt* value of a type, which is the type's default or example value. For instance, the bunt value of `^([@ud @tas])` is `[0 %$]`, which is a cell consisting of the bunt value of `@ud` (`0`) and the bunt value of `@tas` (`%$`). It is the `bunt` function in the specification that generates the bunt value of a type.

```
1 rule <k> ^* \gap P => bunt(P) ... </k>
2 rule <k> ^*(P) => bunt(P) ... </k>
```

Listing 3.17: The rules of `^*` (kettar).

The `^=` (kettis) rune (specified in Listing 3.17) binds names to values (nouns). If more than one name is provided, they are structured in cells of names or as names of names (Listing 3.19 specifies this syntax) and distribute evenly over the noun to name it. For example, `^=(a 1)` produces `a=1` (the name `a` bound to the noun `1`), and `^=([b c d=[x y]] [1 2 3 4])` produces `[b=1 c=2 d=[x=3 y=4]]`. In the specification, this naming procedure is handled by the `name` function.

```
1 rule <k> ^= \gap P \gap Q => name(P, Q) ... </k>
2 rule <k> ^=(P \ace Q) => name(P, Q) ... </k>
```

Listing 3.18: The rules of `^=` (kettis).

```
syntax Skin ::= Name
              | Name "=" Skin
              | "[" Skin "\\ace" Skin "]"
              | "[" Skin "\\ace" Skin "\\ace" SkinCellImplicit "]"
syntax SkinCellImplicit ::= List{Skin, "\\ace"}
```

Listing 3.19: The syntax for skins.

Types in Hoon are actually gates, meaning they operate on a value to coerce it into a certain structure (technically a function from a noun to a noun). For instance, calling the `*` type (the type for any noun) with `[1 2]` as the sample, i.e. `%-(* [1 2])` produces the noun `[1 2]`, but now with the type `*` (any noun) instead of `[@ud @ud]` (a cell of two `@ud`s). This behavior is handled in the specification by building the corresponding gate for a `Spec` in those scenarios where a `Spec` used as a gate.

### 3.3.5 Cores

As discussed previously in Section 2.1.3, a core is a cell of a collection of named Hoon expressions (a battery) encoded as a noun (a Nock formula), and the data on which those computations rely (a payload). To illustrate, consider the battery comprising the expressions `:-(0 1)` and `=>(a .)`, named `a` and `b` respectively. The corresponding Nock formula that encodes this battery is `[[9 5 0 1] 1 0 1]`.

The specification of a battery in KHoon (in its current state) differs somewhat compared to this. Instead of compiling the battery into a Nock formula, the battery is specified as a list of Hoon expressions in KHoon syntax (see Listing 3.20). For instance, the KHoon representation of the battery of the previous example is `:-(0 \ace 1) \sep =>(a \ace \dot) \sep .Battery`. Furthermore, one may have noticed that the names that should be associated with the expressions (`a` and `b`) are not part of the KHoon representation of the battery (or even the actual Nock formula). These are instead part of the metadata that is attached to nouns (see Listings 3.13 and 3.21). This is, the full metanoun representation of the discussed battery example in KHoon is:

```
({*; a, b, .ArmNames}, :-(0 \ace 1) \sep =>(a \ace \dot) \sep .Battery)
```

The reason for this difference arises from a goal of simplification - to ensure that KHoon is self-contained and does not rely on external dependencies. Nevertheless, this method of specifying batteries has a limitation that is examined in Chapter 5, where also potential solutions for representing them accurately are explored.

```
syntax Core ::= "[" Battery "\\ace" Hoon "]" [seqstrict(2)]

syntax Battery ::= List{Hoon, "\\sep"}
```

Listing 3.20: The syntax for cores.

```
syntax ArmNames ::= List{Name, ","}
                  | "[" ArmNames "\\ace" ArmNames "]"
```

Listing 3.21: The syntax for `ArmNames`.

To create cores, bar runes are used. Most of the bar runes expand to the `|%` (barcen) rune, possibly in combination with runes from other families, and as it stands, all currently specified bar runes follow this pattern.

The syntax for the `|%` (barcen) rune is specified in Listing 3.22. It can only be written in tall form and accepts a list of arms. Arm (lus) runes are used to define the arms in a core, but currently, only the `++` (luslus) rune has been specified for this purpose. The rule for the `|%` (barcen) rune is given in Listing 3.23, which formats the named Hoon expressions (the arms) with the subject as the payload, according to the core specification discussed earlier.

```
syntax TallForm ::= "|%" "\\gap" ArmList "\\gap" "--"

syntax Arm ::= "++" "\\gap" Name "\\gap" Hoon
syntax ArmList ::= List{Arm, "\\gap"}
```

Listing 3.22: The syntax for barcen and lus runes.

```
rule <k> |% \gap P \gap -- => buildCore(P, S) ... </k>
     <subject> S->_ </subject>
```

Listing 3.23: The rule of `|%` (barcen).

### 3.3.6 Function Calls

A function call in Hoon evaluates an arm in a core, usually after modifying the sample (the input values provided replace the default sample), and it is the runes belonging to the cen family that are the ones that perform this task. Moreover, the runes in the cen family reduce to the `%=` (centis) rune.

The `%=` (centis) rune resolves a wing of the subject, but with modifications (Listing 3.24 specifies its syntax). The modifications are determined by the list of wing-Hoon pairs provided to the rune, where the noun that the Hoon expression evaluates to replaces the original noun referred to by the wing. For example, Listing 3.25 shows a Hoon expression that makes `[0 a=[b=1 2]]` the subject, and then resolves the wing `a` with `b` changed to `12` (`[b=12 2]` is returned). Additionally, a wing can refer to either an arm or a leg, and the behavior of the `%=` (centis) rune varies depending on which type of wing is being resolved.

```
syntax TallForm ::= "%=" "\\gap" Wing "\\gap" WingHoonTallList
    "\\gap" "=="

syntax WideForm ::= "%=" "(" Wing "\\ace" WingHoonWideList ")"

syntax WingHoonTallPair ::= Wing "\\gap" Hoon
syntax WingHoonTallList ::= List{WingHoonTallPair, "\\gap"}

syntax WingHoonWidePair ::= Wing "\\ace" Hoon
syntax WingHoonWideList ::= List{WingHoonWidePair, ",\\ace"}
```

Listing 3.24: The syntax for `%=` (centis).

```
=>  [0 a=[b=1 2]]
%=  a
  b  12
==
```

Listing 3.25: A `%=` (centis) example.

Listing 3.26 displays the rules governing the behavior of the `%=` (centis) rune. These rules consist of four cases, two for the tall form syntax and two for the wide form syntax, each covering either the arm or the leg case. The semantics of the tall and wide forms are identical, whereby the rune is rewritten to the arm case if the wing represents an arm, and to the leg case otherwise.

```
rule <k> %= \gap P \gap Q \gap == => centisArm(P, Q) ... </k>
     <subject> S->_ </subject> requires isArmName(P, S)
rule <k> %=(P \ace Q) => centisArm(P, Q) ... </k>
     <subject> S->_ </subject> requires isArmName(P, S)
rule <k> %= \gap P \gap Q \gap == => centisLeg(P, Q, .WingHoonTallList)
     ... </k> [owise]
rule <k> %=(P \ace Q) => centisLeg(P, Q, .WingHoonWideList)
     ... </k> [owise]
```

Listing 3.26: The rules of `%=` (centis).

The arm case of `%=` (centis) computes the arm with the modified core as the subject, and the rules that specify this are show in Listing 3.27. They rewrite into three consecutive computations:

1. Modify the parent core (i.e., the core to which the arm belongs) with the provided changes, and push the resulting noun onto the subject stack.
2. Retrieve and evaluate the Hoon expression of the arm.
3. Pop the top of the subject stack, thereby restoring the subject stack to its previous state.

```
syntax Hoon ::= centisArm(Wing, WingHoonTallList)
              | centisArm(Wing, WingHoonWideList)
rule <k> centisArm(P, Q:WingHoonTallList)
     =>  push(applyWingChanges(Q, getParentCore(P, S)))
     ~>  getBatteryCode(P, getBattery(getParentCore(P, S)),
            getCoreArmNames(getParentCore(P, S)))
     ~>  pop() ... </k>
     <subject> S->_ </subject>
rule <k> centisArm(P, Q:WingHoonWideList)
     =>  push(applyWingChanges(Q, getParentCore(P, S)))
     ~>  getBatteryCode(P, getBattery(getParentCore(P, S)),
            getCoreArmNames(getParentCore(P, S)))
     ~>  pop() ... </k>
     <subject> S->_ </subject>
```

Listing 3.27: The specification of the `%=` (centis) arm case.

The leg case of `%=` (centis) modifies the wings of the leg according to the provided changes and returns the resulting modified leg. Listing 3.28 displays the rules that specify this behavior. Similarly to the arm case, these rules also rewrite into three consecutive computations (the third and fourth rules):

1. Push the noun of the leg onto the subject stack.
2. Apply the provided changes.
3. Pop the top of the subject stack, thereby restoring the subject stack to its previous state.

However, before these computations are performed, each Hoon expression in the wing-Hoon pairs is evaluated. That is, the Hoon expressions provided to the `%=` (centis) rune are not evaluated with the leg as the subject. The first and second rules, along with the two context declarations which generate the appropriate heating and cooling rules, evaluate these Hoon expressions prior to performing the three computations described above.

```
syntax Hoon ::= centisLeg(Wing, WingHoonTallList, WingHoonTallList)
              | centisLeg(Wing, WingHoonWideList, WingHoonWideList)
rule <k> centisLeg(P, Q1:WingHoonTallPair \gap Q, R)
     =>  centisLeg(P, Q, Q1 \gap R) ... </k> [owise]
rule <k> centisLeg(P, Q1:WingHoonWidePair ,\ace Q, R)
     =>  centisLeg(P, Q, Q1 ,\ace R) ... </k> [owise]
rule <k> centisLeg(P, .WingHoonTallList, R)
     =>  push({lookup(toWingStack(P, .WingStack), S, .MetaNounStack,
            0)}:>MetaNoun)
     ~>  applyWingChanges(R, {lookup(toWingStack(P, .WingStack), S,
            .MetaNounStack, 0)}:>MetaNoun)
     ~>  pop() ... </k>
     <subject> S->_ </subject>
rule <k> centisLeg(P, .WingHoonWideList, R)
    =>  push({lookup(toWingStack(P, .WingStack), S, .MetaNounStack,
            0)}:>MetaNoun)
    ~>  applyWingChanges(R, {lookup(toWingStack(P, .WingStack), S,
            .MetaNounStack, 0)}:>MetaNoun)
    ~>  pop() ... </k>
     <subject> S->_ </subject>
context centisLeg(_, _ \gap HOLE:Hoon \gap _, _)
context centisLeg(_, _ \ace HOLE:Hoon ,\ace _, _)
```

Listing 3.28: The specification of the `%=` (centis) leg case.

## 3.4 Pretty-Printer

The pretty-printer is a program that extracts the resulting output in the `<k>` cell from executing the semantics and removes the associated metadata. It also replaces `\ace`s with single spaces. This is sufficient because a value in Hoon is always a noun and nouns do not contain gaps or positional address dots. Therefore, gaps (`\gap`) and positional address dots (`\dot`) do not exist here, so there is no need to change them back to their ASCII symbol, assuming the semantics execute successfully.

## 3.5 Testing the Specification

To test our specification, we have developed a test suite. This test suite comprises a collection of small Hoon programs that test specific features or combinations thereof in the specified parts of Hoon. These programs are categorized as either 'good' or 'bad' tests. At present, the test suite encompasses 106 good tests and 25 bad tests, and Figure 3.5 illustrates its directory structure. Good programs are expected to execute successfully and yield a specific result, whereas bad programs are intended to fail. Additionally, the expected result corresponding to a good test is stored in a file with the same name as the test, but with the extension `.output` appended to it. For example, Listing 3.29 and 3.30 show a good test (`tisgar0.hoon`) and its

corresponding expected output (`tisgar0.hoon.output`), while Listing 3.31 shows a bad test that results in a find error.

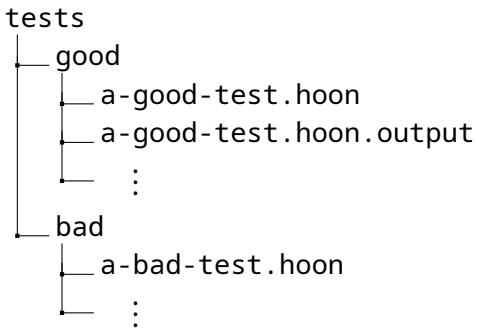



Figure 3.5: The directory structure of the test suite.

```
1 =>  9
2 .
```

Listing 3.29: A good test which makes the subject 9 and then retrieves the subject.

```
1 9
```

Listing 3.30: The expected output of the good test in Listing 3.29.

```
1 =>  [b=a=[a=0 b=1] a=%.y b=%.n b=7]
2 b.a
```

Listing 3.31: A bad test that results in a find error because `b.a` is not in the subject.

The test suite also includes a testing program that executes all the tests. The tests are executed one at a time, following the steps outlined in Figure 3.1. Additionally, the testing program considers a good test passed if the output produced by running the test matches the expected output (stored in the corresponding `.output` file), and considers a bad test rejected if the exit code from running the test is nonzero.

# 4

# Deductive Verification

In this chapter, we will present and prove several claims concerning our $\mathbb{K}$ specification of Hoon, using the theorem prover in the $\mathbb{K}$ framework. We will begin by discussing how to formulate and prove basic claims. Subsequently, we will examine a claim pertaining to the example program provided in Listing 2.1, albeit in a slightly simplified form. Additionally, we will present claims relating to the gates found in the Urbit kernel code.

## 4.1 Verification of Basic Claims

To start with a simple example, consider the claim shown in Listing 4.1. This claim asserts that the `:-` (colhep) rune, which produces a cell of its two children, produces the cell `[0 1]` with type `[@ud @ud]` given `0` and `1`. There are a few important points to address in this claim. Firstly, the claim is named 'cell.' Secondly, the arm names metadata is disregarded using `?_`. Thirdly, `@ud`s are written using the `String` to `UD` conversion to distinguish them from `Int`s. This distinction is crucial because without it, a parse error occurs when the claim is given to the theorem prover. The parser interprets `0` and `1` as belonging to the `Int` sort instead of `UD`, and the `:-` (colhep) rune expects children of the `Hoon` sort (`Int` is not a part of the `Hoon` sort).

```
1 claim [cell]:
2       <k>
3         :-(StringToUD("0") \ace StringToUD("1"))
4       =>
5         ({[@ud \ace @ud]; ?_}, [StringToUD("0") \ace StringToUD("1")]:Cell)
6         ...
7       </k>
```

Listing 4.1: A claim stating that `:-(0 1)` rewrites to the cell `[0 1]` with type `[@ud @ud]`.

When this claim is provided to the theorem prover, it returns the result `#Top` after approximately 6 seconds[1], indicating that the claim has been successfully proven.

[1]This is on a machine with an Intel i7-7700K (8) @ 4.500GHz and an NVIDIA GeForce GTX 1080, and the same applies to all subsequent runtimes mentioned in this chapter. Furthermore, Table C.2 contains all of these runtimes.

This is a rather straightforward example of how to formulate and prove a claim about KHoon using the $\mathbb{K}$ framework. To make things more interesting, the next claim (in Listing 4.2) is a bit more sophisticated.

The claim in Listing 4.2 contains a KHoon program (lines 3-7) that gives the arbitrary `@ud A` the name `a`, as well as giving the arbitrary `@ud B` the name `b`, and combines them with the subject. It then names the resulting noun from evaluating the expression on line 6 as `c` and combines it with the subject. The expression on line 6 evaluates to the value of `a` if the value of `a` is equal to the value of `b` plus one; otherwise, it returns the value of `b`. Finally, the value of `c` is returned.

```
1  claim [AorB0]:
2        <k>
3          =/ \gap a \gap A:UD \gap
4          =/ \gap b \gap B:UD \gap
5          =/ \gap c \gap
6            ?:(.=(a \ace .+(b)) \ace a \ace b) \gap
7          c
8        =>
9          ({@ud; ?_}, ?C)
10         ...
11       </k>
12       <subject> _->.Subject </subject>
13       ensures ?C ==K A orBool ?C ==K B
```

Listing 4.2: A claim that the resulting `@ud` from evaluating a Hoon expression, which returns `A` if `A` is equal to `B` plus 1, otherwise `B`, will be either `A` or `B`.

The claim asserts that this program is rewritten to a metanoun with type `@ud` and some resulting value `?C`. It also asserts, with the ensure clause, that this value is equal to either `A` or `B`. Moreover, a requirement of the claim is that the `<subject>` cell is a nonempty stack of metanouns. This is crucial because if the stack was allowed to be empty, or in other words, if the subject was undefined, the program would fail. Specifically, the `=/` (tisfas) rune eventually rewrites to a `\dot` (among other things), which is a positional address used to retrieve the current subject (a noun). If the subject stack is empty, there is no noun to retrieve, and the program cannot continue to evaluate, causing it to fail. Therefore, it is necessary that the subject stack be nonempty. This reasoning about addressing the subject also applies to later claims that contain the same `<subject>` cell.

As before, the theorem prover can prove this claim as it is, with an approximate runtime of 9 seconds. However, this is not the case for the next claim in Listing 4.3. Although this claim is essentially the same as the previous one, the program is now expressed as a gate and invoked with a cen rune, along with a sample that includes the arbitrary `@ud`s, `A`, and `B`. While attempting to prove this claim, the theorem prover gets stuck because some of the associated constraints cannot be proven.

A solution to this problem is to add the simplification rule defined in Listing 4.4. This rule specifies that converting a token with the `UD` sort to an `Int` and then back

to `UD` is equivalent to the original `UD` token. For instance, the result of converting the `UD` token `12.345` to an `Int` is `12345`, and then back to an `UD` again is `12.345` (the only difference is removing and inserting the thousands separator).

```
claim [AorB1]:
      <k>
        =< \gap
        %-(foo \ace [A:UD \ace B:UD]:Cell) \gap
        ^= \gap foo \gap
        |= \gap [a=@ud \ace b=@ud] \gap
        ^- \gap @ \gap
        =/ \gap c \gap
          ?:(.=(a \ace .+(b)) \ace a \ace b) \gap
        c
      =>
        ({@; ?_}, ?C)
        ...
      </k>
      <subject> _->.Subject </subject>
      ensures ?C ==K A orBool ?C ==K B
```

Listing 4.3: The same claim in Listing 4.2 but written and called as a gate. The theorem prover proves this in approximately 13 seconds, as opposed to the original runtime of 9 seconds.

```
rule StringToUD(IntStringToUDString(Int2String(String2Int(replace(
    UDToString(A), ".", "", countAllOccurrences(UDToString(A), "."
    ))))) => A [simplification]
```

Listing 4.4: A simplification rule that states that converting a token of the `UD` sort to an `Int`, and then back to an `UD` is equivalent to the original `UD`.

This simplification rule is derived based on the output from the theorem prover. In this case, the output includes information from the final configuration as well as some negative and positive matching logic constraints. The rule is derived by utilizing the information from the final configuration and the positive constraints to falsify one of the negative constraints.

## 4.2 Verification of a Summation Loop

As an example of a simple program to prove that it conforms to the example in Section 1.1 (the Goal section), the claim in Listing 4.5 is made. This claim contains a program that adds the numbers from one to five, and the program is the same as the one in Listing 2.1, but it has been simplified and modified in the following ways (refer to Appendix B for the Hoon version):

- The program now checks for equality with 6 instead of checking for values greater than 5.

- Most things have been rewritten from their irregular form to their regular form. For example, `(add counter 1)` has been rewritten as `%-(add [counter 1])`.
- The `add` and `dec` gates have been explicitly defined and combined with the subject.

Furthermore, the claim asserts that the program rewrites to 15, and the theorem prover can prove this directly in approximately 12 minutes.

The reason for these changes is because the specification is incomplete. Not all runes or irregular forms are specified, hence the simplifications. Additionally, nothing is assumed to already be in the subject before execution, hence the explicit addition of gates to the subject.

```
1  claim [sum]:
2        <k>
3          =< \gap
4          =/ \gap counter \gap StringToUD("1") \gap
5          =/ \gap sum \gap StringToUD("0") \gap
6          |- \gap
7          ?: \gap .=(counter \ace StringToUD("6")) \gap
8            sum \gap
9          %= \gap $ \gap
10           counter \gap %-(add \ace [counter \ace StringToUD("1")]:Cell) \gap
11           sum \gap %-(add \ace [sum \ace counter]:Cell) \gap
12         == \gap
13         |% \gap
14         ++ \gap add \gap
15           |= \gap [a=@\aceb=@] \gap
16           ^- \gap @ \gap
17           ?: \gap .=(StringToUD("0") \ace a) \gap b \gap
18           %=($ \ace a \ace %-(dec \ace a),\ace b \ace .+(b)) \gap
19         ++ \gap dec \gap
20           |= \gap a=@ \gap
21           ?< \gap .=(StringToUD("0") \ace a) \gap
22           =+ \gap b=StringToUD("0") \gap
23           |- \gap ^- \gap @ \gap
24           ?: \gap .=(a \ace .+(b)) \gap b \gap
25           %=($ \ace b \ace .+(b)) \gap
26         --
27       =>
28         ({@; ?_}, StringToUD("15"))
29         ...
30       </k>
31       <subject> _->.Subject </subject>
```

Listing 4.5: A claim that the program in B.1 rewrites to 15.

## 4.3 Verification of Urbit Kernel Code Gates

This section verifies a claim about the `dec` gate and discusses how similar claims about the `add`, `sub`, and `mul` gates can be verified. The implementation of these gates is copied from the Urbit kernel code but with the ~ (sig) runes removed. Since these runes only pass non-semantic information to the interpreter, removing them does not affect the functionality of the gate. The reason for their removal is that they have not yet been specified.

The `dec` gate performs an unsigned decrement operation on its sample, which must be an atom. Moreover, in case the sample is zero, an error is thrown. A claim regarding this gate is presented in Listing 4.6. It asserts that calling the `dec` gate with an arbitrary `@ud` value greater than 0 results in a value that is one less than the initially provided `@ud` value.

```
1  claim [dec]:
2        <k>
3          =< \gap
4          %-(dec \ace A:UD) \gap
5          |% \gap
6          ++ \gap dec \gap
7            |= \gap a=@ \gap
8            ?< \gap .=(StringToUD("0") \ace a) \gap
9            =+ \gap b=StringToUD("0") \gap
10           |- \gap ^- \gap @ \gap
11           ?: \gap .=(a \ace .+(b)) \gap b \gap
12           %=($ \ace b \ace .+(b)) \gap
13         --
14       =>
15         ({@; ?_}, ?B)
16         ...
17       </k>
18       <subject> _->.Subject </subject>
19       requires UDToInt(A) >Int 0
20       ensures  UDToInt(?B) ==Int UDToInt(A) -Int 1
```

Listing 4.6: A claim asserting that calling the `dec` gate with an arbitrary `@ud` value greater than 0 results in a value that is one less than the initially provided `@ud` value.

When given this claim, the theorem prover successfully proves, one positive integer at a time, that the claim is true, resulting in an infinite proof. Figure 4.1 depicts a tree graph visualization of this proof process, which shows the initial 178 steps of the proof. In the figure, a green outline on a leaf indicates that the branch has been proven, and each node that branches into two checks whether the provided atom is equal to one, two, three, and so on. The leftward branches correspond to the scenario in which this is true.

By specifying an invariant about the expression invloving the `%=` (centis) rune on line 12 in Listing 4.6, the theorem prover can prove that the claim holds for all `@ud`

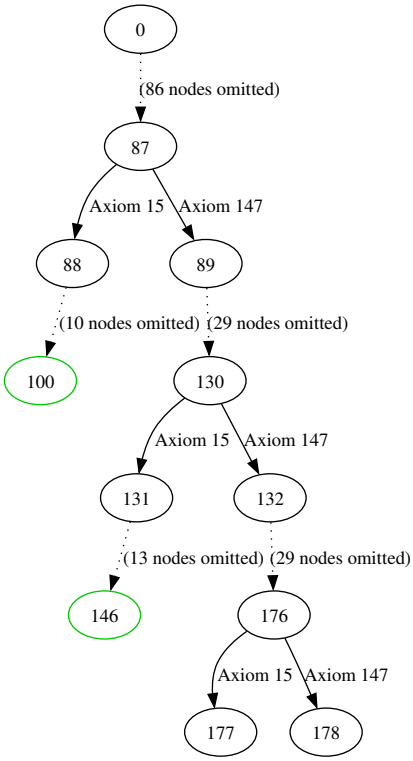



Figure 4.1: The first 178 steps of proving the claim in Listing 4.6.

values (proof by induction). This invariant is partially shown in Listing 4.7, and it asserts that the expression on line 12 in Listing 4.6 rewrites to the value `A` minus one, assuming that `A` is greater than `B` plus one (`A` and `B` are in the subject). It also covers the cases where the type (aura) of `B` is either `@ud` or `@`. The reason for this is because the first time the `%=` (centis) rune is reached, `B` has been inferred to be of the `@ud` aura, but in subsequent iterations, it has been cast to the more general `@` aura by the `^-` (kethep) rune on line 10. Hence, in order for the invariant to match both these scenarios, the aura of `B` can be either `@ud` or `@`.

```
1  claim [dec-loop-inv0]:
2        <k>
3          %= ( $ \ace b \ace .+ ( b ) )
4        =>
5          ({@; ?_}, IntToUD(UDToInt(A) -Int 1))
6          ...
7        </k>
8        <subject>
9          // Intentionally Omitted
10       </subject>
11       requires (T ==K @ud orBool T ==K @) andBool
12                UDToInt(A) >Int UDToInt(B) +Int 1 [trusted]
```

Listing 4.7: A loop invariant for the 'dec' claim with the `trusted` attribute.

The content of the `<subject>` cell in the claim is not included in the listing due to its large size. The `<subject>` cell consists of a metanoun located at the top of the stack, which matches the subject on line 12 and is followed by an arbitrary stack of metanouns. The reason for specifying only the metanoun at the top of the stack is to ensure that the invariant matches at each iteration. In each iteration, a new metanoun is pushed onto the subject stack with the value of `b` being incremented by one. Therefore, specifying the `<subject>` cell to be exactly the same as any iteration would not work (the `<subject>` cell is different in each iteration).

By adding the trusted attribute to this claim, it is automatically included in the list of proven things, eliminating the need for separate proof. Moreover, by specifying a few simplification rules and providing the trusted invariant along with the claim in Listing 4.6, the theorem prover is able to complete the proof in approximately 12 seconds (compare Figure 4.2 with 4.1).

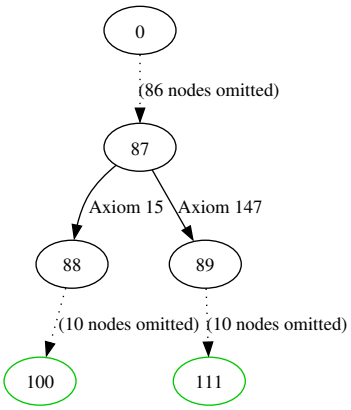
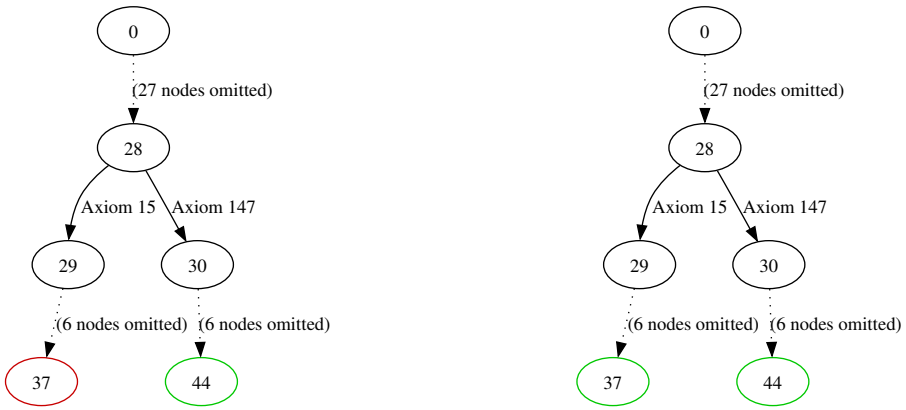



Figure 4.2: The proof steps to prove the 'dec' claim using the invariant in Listing 4.7.

If the `trusted` attribute is removed and the claim is given to the theorem prover, the proof fails due to a specific issue in the base case. In Figure 4.3a, the red outlined leaf in the left branch, represents the problematic state. The problem arises because the content of the `<k>` cell rewrites to `IntToUD(UDToInt(B) +Int 1)` instead of `IntToUD(UDToInt(A) -Int 1)`, as stated in the claim. This discrepancy causes the theorem prover to get stuck at this point. Interestingly, the stuck state involves a positive ML constraint, indicating that `IntToUD(UDToInt(A) +Int -1)` is equal to `IntToUD(UDToInt(B) +Int 1)`. Despite this constraint, the proof does not terminate successfully. Perhaps it might be possible to write a simplification rule to resolve this issue.

An approach that does resolve the issue, however, is to modify the invariant as shown in Listing 4.8. In this revised version, the only difference is the utilization of the `ensures` clause and the conversion of the result to an `Int`. When the theorem prover is presented with this modified invariant, it is able to successfully prove it within approximately 27 seconds, as illustrated in Figure 4.3b. This indicates that the revised invariant effectively addresses the previous proof failure.




(a) The proof steps for the claim in Listing 4.7.

(b) The proof steps for the claim in Listing 4.8.

Figure 4.3: The proof steps for the two different versions of the invariant regarding the 'dec' claim.

```
1  claim [dec-loop-inv1]:
2        <k>
3          %= ( $ \ace b \ace .+ ( b ) )
4        =>
5          ({@; ?_}, ?C)
6          ...
7        </k>
8        <subject>
9          // Intentionally Omitted
10       </subject>
11       requires (T ==K @ud orBool T ==K @) andBool
12                UDToInt(A) >Int UDToInt(B) +Int 1
13       ensures  UDToInt(?C) ==Int UDToInt(A) -Int 1
```

Listing 4.8: Another version of the claim in Listing 4.7, which omits the `trusted` attribute and is written using the `ensures` clause, where the result is converted to an `Int`.

However, this version of the invariant encounters a different problem when applied to the 'dec' claim. When attempting to prove the 'dec' claim using this modified invariant, the theorem prover cannot proceed and throws some kind of error at the step where the invariant applies. To understand the cause of this error and determine the appropriate steps to address it, further investigation and analysis are required.

Regarding the claims of the `add`, `sub`, and `mul` gates mentioned earlier in this section (refer to Appendix D), a preliminary examination suggests that the proving process for these gates closely resembles that of the 'dec' claim. When subjected to a theorem prover, it initiates a similar proof process as depicted in Figure 4.1, which may involve some potentially infeasible branches. However, these infeasible branches can likely be eliminated by deducing and specifying appropriate simplification rules.

As such, it should only be a matter of formulating the corresponding invariants for these claims, in comparison to the `dec` invariant, in order to properly prove them.

# 5

# Discussion and Conclusion

This chapter elaborates on the project's results, which include a review of the formal specification, outcomes of the deductive verification, and an analysis of the test suite. It also examines potential solutions to current problems, highlights possible directions for future work, and presents some related work to provide context and insights into the project's broader field of research.

## 5.1 Formal Specification

The current specification falls short of covering the entire Hoon language, leaving some portions unspecified. To put it into perspective, the specification fully or partially specifies only 32 out of a total of 127 runes, which amounts to approximately 25% of the runes, along with wings and three kinds of atoms, all accomplished with roughly 1000 lines of $\mathbb{K}$ code (see Appendix C for more details). Furthermore, in addition to this partial achievement, the current specification also has several issues that need to be addressed.

Regarding the incompleteness of the specification, both non-rune and rune expressions have missing parts. Specifically, two non-rune expressions, path with interpolation and text string with interpolation, remain completely unspecified, and most atom kinds lack specification. To address this, the specification needs to include the missing non-rune expressions, along with their required functionality. This will involve figuring out how to specify them and then adding them to the specification. As for the remaining atom kinds, adding them should be a matter of following the same process used to specify the current atoms.

In terms of the runes, some of them would require completely new functionality, while others simply expand either fully or partially to existing runes in the specification. The former mostly applies to the runes in completely unspecified rune families. Including them, along with their required functionality, in the specification will involve figuring out how to specify them and then adding them, similarly to the completely unspecified non-rune expressions. The latter, on the other hand, mostly applies to the rune families that have been partially specified. Adding these should be straightforward since most of the infrastructure is already in place. Therefore, increasing the number of specified runes, whether fully or partially, by addressing these runes should be fairly easy.

Concerning the issues with the current specification, they involve the output produced by executing the semantics, the handling of types and errors, and the representation of cores. The issue with the output concerns writing cells with omitted brackets. According to the specification, every cell in the input program with omitted brackets is converted to its corresponding form, which includes all brackets. When the semantics execute a Hoon program, the resulting value (noun) will still have all brackets explicitly. Although this is not incorrect, it will not be pretty-printed as expected. For example, if the result of running a Hoon program is `[0 1 2 3 4 5]`, the specification will return `[0 [1 [2 [3 [4 5]]]]]`. As a result, the test suite will report that some of the good tests do not pass, even though they technically do.

To address this problem, one could make the pretty-printer more sophisticated by adding the functionality to remove such brackets in cells. However, an easier solution might be to write rules in the specification that remove these brackets once the resulting value has been computed, for instance, by moving it to a cell where these rules apply.

Typing has a limitation in that it only accommodates nouns and does not provide support for type unions. The latter limitation arises because the specification does not include the `$?` (bucwut) rune, which is used to form a type from a union of other types. In Hoon, typing should be possible for any expression, not just nouns. For instance, consider the Hoon expression `?:(%.y %foo 0)`. Its type should be a union of `%foo` and `@ud`. The `?:` (wutcol) construct in Hoon acts as an if-else statement, and its type should be the union of the types of its true and false branches. To enable this broader typing functionality, it would be necessary to extend the type inference capabilities of the specification.

Error handling has not been specified at all. The only time an error is thrown during the execution of the semantics is when the generated interpreter encounters an error. For instance, this can occur when the syntax of the interpreted file does not match the syntax specified. However, certain expressions should trigger a specific type of error, such as a nest-fail error. For example, when evaluating the expression `%-(dec [0 0])`, a nest-fail error should occur since the `dec` gate requires an atom, not a cell. To resolve this, error handling needs to be added to the specification.

The problem with the representation of cores in the specification is that the battery is currently a list of Hoon expressions instead of being compiled into a Nock formula, which is a noun. While this may not pose a problem when simply running code in a core, it becomes problematic when using addressing that prints the contents of the battery. In such cases, the representation is incorrect because it should be a Nock formula, but it is not.

To obtain a Nock formula in the battery, there are several possible approaches. One approach is to make a system call to an external compiler and read the result when needed. Another approach is to statically identify all the units of code that need to be compiled to Nock, precompile them, write them to a file, and load them when needed.

Overall, progress has been achieved in the formalization of a semantics for Hoon,

allowing for the successful execution of programs such as the one presented in Appendix B. Nonetheless, it is important to acknowledge that the current semantics remain incomplete, and certain issues still need to be addressed. Hence, further work is required to refine and expand the specification.

## 5.2 Test Suite

The goal of the test suite was to comprehensively assess the robustness of the specification by executing programs that perform special cases of features and their combinations. Progress has been made towards this goal, with a total of 131 tests in the suite. Each non-rune and rune expression specified currently has at least one dedicated test, and many have multiple tests to account for various combinations of different features. Furthermore, the test suite includes a program that runs all the tests on the specification.

Although every possible feature or combination of features was attempted to be covered in the test suite, there is still a chance that some cases may have been missed, especially with the addition of new features to the specification. To address this potential issue, one may spend more time examining and developing the test suite further to ensure that all scenarios are properly covered. By taking a comprehensive approach, it should be possible to mitigate this potential problem and ensure the reliability of the test suite.

One limitation of the current implementation of the test suite is its handling of errors, specifically the condition that a bad program is rejected. Currently, the test suite only checks if processing the input Hoon program as illustrated in Figure 3.1 results in a nonzero exit code, and considers it rejected if that is the case. Therefore, it cannot differentiate between different types of errors and cannot determine if the test was rejected for the right reason, such as a syntax or a type error. However, once the specification includes proper error handling, it should be straightforward to extend the tester program with this functionality.

Overall, the test suite comprises 131 tests and provides a certain degree of assurance that the specified parts of the Hoon are accurate. Although it may not guarantee complete accuracy, the comprehensiveness of the test suite, as well as its coverage of a wide range of scenarios, gives some confidence in the accuracy of the specification.

## 5.3 Deductive Verification

The theorem prover provided by the $\mathbb{K}$ framework is able to prove claims about our partial specification of Hoon in $\mathbb{K}$ as demonstrated in Chapter 4. Furthermore, during the course of proving such claims, issues within the specification were identified and subsequently addressed. Nonetheless, certain inconveniences presently exist with regard to formulating claims and undertaking their verification.

An example of an issue with the specification that was discovered and resolved by attempting to prove claims was related to overlapping rules. In certain situations,

multiple rules could be applied to the current configuration, where the application of some of these rules eventually led back to the same configuration. Furthermore, the branches created by some of these rules generated more branches with the same property, causing a sequence of steps to repeat and branch in an infinite and exponential manner. As a result, the prover threw an out-of-memory error once the memory was exhausted. By utilizing the debugging feature of the theorem prover, these rules were identified, and appropriate priority values were assigned to prevent further overlap.

Regarding the inconveniences, they all seem to revolve around the use of both the `UD` and `Int` sorts. For example, one inconvenience is the need to convert from `String` to `UD` in order to write numbers of the `UD` sort in claims. Another inconvenience is the requirement to specify trivial simplification rules for the conversions between `Int` and `UD` when proving claims. Furthermore, the use of both the `Int` and `UD` sorts may contribute to the issues regarding the invariant discussed in Section 4.3. The reasoning for this is that the second version of the invariant, which converts the result to an `Int` in the ensures clause, is provable, whereas the first version, which does not perform this conversion, currently fails.

To avoid these inconveniences, it might be better to simply use the `Int` sort exclusively for integers in the specification. The preparser and pretty-printer can be extended with the functionality to convert the different integer syntaxes in Hoon to and from $\mathbb{K}$ `Int`s.

Overall, several claims about the specifications have been verified, and the goal to start verifying simple programs has been met.

## 5.4 Related Work

$\mathbb{K}$ has already been used as a semantic framework and development tool for several formal semantics of real-world programming languages. Two such examples are KJS and K-Java.

KJS is a formal semantics of JavaScript in $\mathbb{K}$ [2]. Prior to KJS, efforts to provide formal semantics to JavaScript had been made, but they were all problematic in some way, such as being incomplete or containing errors. Furthermore, different purposes required different formalizations, such as having one semantics for execution and another for deductive reasoning, which is impractical considering equivalence proofs and their maintenance with language updates. To solve these problems, KJS was introduced as a single semantics that serves as a reference model for the language and can also be used to develop tools for formal analysis.

Formalizing a semantics for JavaScript presented several challenges. Not only does the language contain many corner cases and various ambiguities, but it is also complex and non-deterministic. Despite these challenges, the paper presents a complete semantics, passing all core language tests in the test262 ECMAScript 5.1 test suite, and took only four months to develop for a first-year PhD student, without prior knowledge of JavaScript or $\mathbb{K}$.

K-Java is another formal semantics in $\mathbb{K}$, but for Java [3]. As with JavaScript, attempts to provide a formal semantics for Java had been made before K-Java, but they were far from complete or well-tested. K-Java, on the other hand, contributes both a complete semantics for Java 1.4 and a comprehensive test suite covering the Java constructs. This was achieved despite the challenges posed by Java, including its large size and numerous corner cases associated with each new feature.

## 5.5 Future Work

The current specification suffers from incompleteness and contains multiple issues that need to be addressed. Additionally, there is significant potential for expanding the verification aspect of the project, particularly as the specification coverage increases. Hence, to further develop this project, the issues discussed in this chapter could be addressed, and the coverage of Hoon specifications could be expanded to encapsulate the entire language. Moreover, additional properties about the language and Hoon programs could be verified.

## 5.6 Conclusion

This project has explored how a formal semantics for Hoon can be specified in the $\mathbb{K}$ framework. This involved performing a preparse and pretty-print run on the input Hoon program and the output Hoon value, respectively. The specification involved writing formal syntax and semantics with a configuration of two cells: the `<k>` cell, which contains the Hoon computation to perform (initially the Hoon program), and the `<subject>` cell, which contains a stack of nouns, where the noun at the top is the subject associated with the top of the `<k>` cell. This way of specifying Hoon has not been problematic in itself, despite the current issues with the specification arising from its incompleteness. Moreover, with this specification, we have shown that it is possible to write claims about simple Hoon programs and prove them using $\mathbb{K}$'s theorem prover. This capability demonstrates the potential of the framework in supporting formal reasoning about Hoon programs. However, there are currently some inconveniences complicating this process. Further work is required to complete the specification and enable its potential future applications.

# Bibliography


[1] G. Roșu and T. F. Șerbănuță, "An overview of the K semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[2] D. Park, A. Stefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 346–356.

[3] D. Bogdanas and G. Roşu, "K-Java: A complete semantics of Java," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 445–456.

[4] *Hoon*, Accessed: 2023-05-23. [Online]. Available: `https://developers.urbit.org/reference/hoon`.

[5] *Arvo*, Accessed: 2023-05-23. [Online]. Available: `https://developers.urbit.org/reference/arvo`.

[6] C. Yarvin, P. Monk, A. Dyudin, and R. Pasco, *Urbit: A solid-state interpreter*, 2016.

[7] *Uqbar Clearpaper*, Accessed: 2023-05-23. [Online]. Available: `https://uqbar-network.gitbook.io/uqbar-clearpaper/uqbar-clearpaper/uqbar-clearpaper`.

[8] *Nock*, Accessed: 2023-05-23. [Online]. Available: `https://developers.urbit.org/reference/nock`.

[9] *K user manual*, Accessed: 2023-05-23. [Online]. Available: `https://kframework.org/docs/user_manual/`.

[10] *Basic builtin types in K*, Accessed: 2023-05-23. [Online]. Available: `https://kframework.org/k-distribution/include/kframework/builtin/domains/`.

[11] X. Chen and G. Roşu, "A language-independent program verification framework," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II 8*, Springer, 2018, pp. 92–102.

# A

# Pronouncing Hoon

Since there is no standard way of pronouncing the ASCII characters used in Hoon (in the runes mostly), there is a convention that provides a unique way to refer to each one (shown in Table A.1). Each name is a monosyllabic mnemonic, and to pronounce a rune one simply combines the names of the ASCII characters that compose it. For example, the rune `=/` is called "tisfas" and the rune `|=` is called "bartis".

Table A.1: A convention for how to uniquely refer to the symbols used in Hoon.

| Name | Character | Name | Character | Name | Character |
|---|---|---|---|---|---|
| ace | □ | gap | □□ \n | pat | @ |
| bar | \| | gar | > | sel | [ |
| bas | \ | hax | # | ser | ] |
| buc | $ | hep | - | sig | ~ |
| cab | _ | kel | { | soq | ' |
| cen | % | ker | } | tar | * |
| col | : | ket | ^ | tic | ` |
| com | , | lus | + | tis | = |
| doq | " | mic | ; | wut | ? |
| dot | . | pal | ( | zap | ! |
| fas | / | pam | & | gal | < |
| par | ) | | | | |

# B

# An Alternative Version of the Hoon Program in Listing 2.1

```
=<
=/  counter  1
=/  sum  0
|-
?:  .=(counter 6)
  sum
%=  $
  counter  %-(add [counter 1])
  sum      %-(add [sum counter])
==
|%
++  add
  |=  [a=@ b=@]
  ^-  @
  ?:  .=(0 a)  b
  %=($ a %-(dec a), b .+(b))
++  dec
  |=  a=@
  ?<  .=(0 a)
  =+  b=0
  |-  ^-  @
  ?:  .=(a .+(b))  b
  %=($ b .+(b))
--
```

Listing B.1: A Hoon program that computes and returns the sum $1+2+3+4+5$. It is written mostly without any irregular forms and does not require anything to be in the subject before evaluating the code.

This program is the same as the one in Listing 2.1, but it has been simplified and modified in the following ways:

1. The program now checks for equality with 6 instead of checking for values greater than 5.
2. Most things have been rewritten from their irregular form to their regular form. For example, `(add counter 1)` has been rewritten as `%-(add [counter 1])`.

3. The `add` and `dec` gates have been explicitly defined and combined with the subject.

The respective reason for these changes is due to:

1. The `gth` gate requires runes that are not currently not part of the specification.
2. The current specification has limited support for irregular forms.
3. The program assumes that no gate is already present in the subject.

# C

# Tables Containing Various Information Regarding the Project

The following tables provide information regarding the lines of code associated with each component of the project, as well as the runtime[1] required to prove the claims about KHoon that have been proven in this paper.

Table C.1: The number of lines of code for each component of the project.

| Component | Lines of Code |
|---|---|
| Preparser | 44 |
| Specification | 947 |
| Pretty-printer | 36 |
| Test suite | 862[2] |
| Verification file | 277 |

Table C.2: The runtime required to prove each KHoon claim that has been proven in this paper.

| Claim(s) | Runtime |
|---|---|
| `cell` | ~ 6 seconds |
| `AorB0` | ~ 9 seconds |
| `AorB1` | ~ 13 seconds |
| `sum` | ~ 12 minutes |
| `dec`, `dec-loop-inv0`[3] | ~ 12 seconds |
| `dec-loop-inv1` | ~ 27 seconds |

[1] on a machine with an Intel i7-7700K (8) @ 4.500GHz and an NVIDIA GeForce GTX 1080
[2] This includes the tester program and all of the tests.
[3] The `dec` claim with the trusted loop invariant.

# D

# Claims Regarding the `add`, `sub`, and `mul` Gates

The claims in Listings D.1, D.2, and D.3 are the corresponding claims for the `add`, `sub`, and `mul` gates, respectively, as compared to the `dec` claim in Listing 4.6. These three gates perform unsigned addition (`add`), unsigned subtraction (`sub`), and unsigned multiplication (`mul`).

```
claim [add]:
      <k>
        =< \gap
        %-(add \ace [A:UD \ace B:UD]:Cell) \gap
        |% \gap
        ++ \gap add \gap
          |= \gap [a=@\aceb=@] \gap
          ^- \gap @ \gap
          ?: \gap .=(StringToUD("0") \ace a) \gap b \gap
          %=($ \ace a \ace %-(dec \ace a),\ace b \ace .+(b)) \gap
        ++ \gap dec \gap
          |= \gap a=@ \gap
          ?< \gap .=(StringToUD("0") \ace a) \gap
          =+ \gap b=StringToUD("0") \gap
          |- \gap ^- \gap @ \gap
          ?: \gap .=(a \ace .+(b)) \gap b \gap
          %=($ \ace b \ace .+(b)) \gap
        --
      =>
        ({@; ?_}, ?C)
        ...
      </k>
      <subject> _->.Subject </subject>
      ensures UDToInt(?C) ==Int UDToInt(A) +Int UDToInt(B)
```

Listing D.1: A claim asserting that calling the `add` gate with two arbitrary `@ud` values results in a value that is the sum of the two initially provided `@ud` values.

```
claim [sub]:
      <k>
        =< \gap
        %-(sub \ace [A:UD \ace B:UD]:Cell) \gap
        |% \gap
        ++ \gap sub \gap
          |= \gap [a=@ \ace b=@] \gap
          ^- \gap @ \gap
          ?: \gap .=(StringToUD("0") \ace b) \gap a \gap
          %=($ \ace a \ace %-(dec \ace a),\ace b \ace %-(dec \ace b)) \gap
        ++ \gap dec \gap
          |= \gap a=@ \gap
          ?< \gap .=(StringToUD("0") \ace a) \gap
          =+ \gap b=StringToUD("0") \gap
          |- \gap ^- \gap @ \gap
          ?: \gap .=(a \ace .+(b)) \gap b \gap
          %=($ \ace b \ace .+(b)) \gap
        --
      =>
        ({@; ?_}, ?C)
        ...
      </k>
      <subject> _->.Subject </subject>
      requires UDToInt(A) >=Int UDToInt(B)
      ensures  UDToInt(?C) ==Int UDToInt(A) -Int UDToInt(B)
```

Listing D.2: A claim asserting that calling the sub gate with two arbitrary @ud values, where the first value is greater than or equal to the second, results in a value that is the difference between the two initially provided @ud values.

```
claim [mul]:
      <k>
        =< \gap
        %-(mul \ace [A:UD \ace B:UD]:Cell) \gap
        |% \gap
        ++ \gap mul \gap
          |: \gap [^=(a \ace ^-(@ \ace StringToUD("1"))) \ace ^=(b \ace ^-(@
               \ace StringToUD("1")))]:Cell \gap
          ^- \gap @ \gap
          =+ \gap c=StringToUD("0") \gap
          |- \gap
          ?: \gap .=(StringToUD("0") \ace a) \gap c \gap
          %=($ \ace a \ace %-(dec \ace a),\ace c \ace %-(add \ace [b \ace c]
              :Cell)) \gap
        ++ \gap add \gap
          |= \gap [a=@\aceb=@] \gap
          ^- \gap @ \gap
          ?: \gap .=(StringToUD("0") \ace a) \gap b \gap
          %=($ \ace a \ace %-(dec \ace a),\ace b \ace .+(b)) \gap
        ++ \gap dec \gap
          |= \gap a=@ \gap
          ?< \gap .=(StringToUD("0") \ace a) \gap
          =+ \gap b=StringToUD("0") \gap
          |- \gap ^- \gap @ \gap
          ?: \gap .=(a \ace .+(b)) \gap b \gap
          %=($ \ace b \ace .+(b)) \gap
        --
      =>
        ({@; ?_}, ?C)
        ...
      </k>
      <subject> _->.Subject </subject>
      ensures UDToInt(?C) ==Int UDToInt(A) *Int UDToInt(B)
```

Listing D.3: A claim asserting that calling the `mul` gate with two arbitrary `@ud` values results in a value that is the product of the two initially provided `@ud` values.