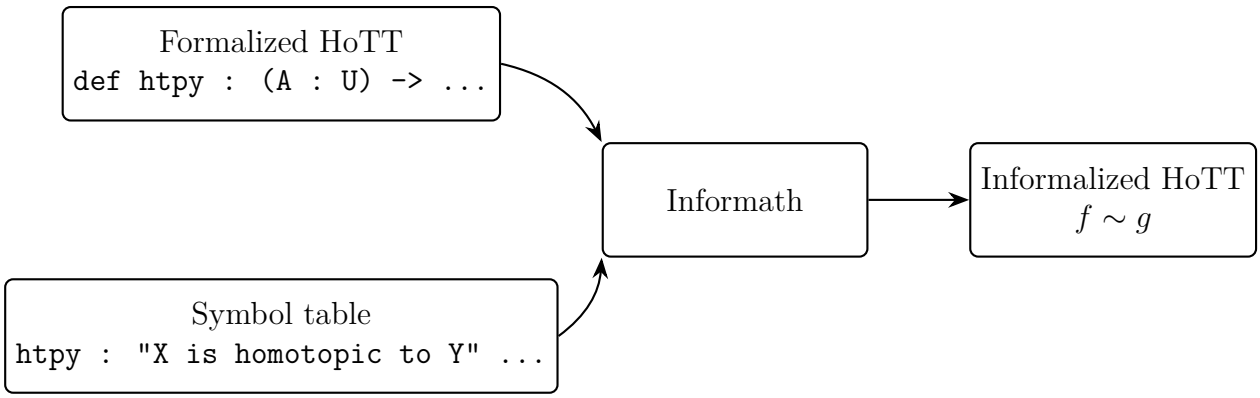
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Extending the Informalization of Mathematics to Homotopy Type Theory

Master's thesis in Computer Science and Engineering

May Ohlsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Master's thesis 2026

Extending the Informalization of Mathematics to Homotopy Type Theory

May Ohlsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Extending the Informalization of Mathematics to Homotopy Type Theory

May Ohlsson

Supervisor: Aarne Ranta, Department of Computer Science and Engineering
Examiner: Ana Bove, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A diagram showing how a Dedukti theory and a symbol table are the inputs required to create an Informath informalization. The image is generated in TikZ.

Typeset in LaTeX
Gothenburg, Sweden 2026

Extending the Informalization of Mathematics to Homotopy Type Theory

May Ohlsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The informalization of mathematics, i.e., turning formal mathematical expressions into informal natural language, is a developing area of study. Probabilistic approaches based on large language models have been used in earlier works to accomplish this task. Informath is an ongoing project to enable deterministic translation between formal and informal mathematics in multiple languages. It is built on Grammatical Framework and formalized Dedukti theory. It has not been demonstrated how this approach generalizes to advanced mathematics. We show how Informath can generate informal representations of the mathematics of Homotopy Type Theory. This is done by formalizing the book *Introduction to Homotopy Type Theory* by Egbert Rijke and extending the grammar to enable generation of informal expressions. We found that, while the theory could be formalized and informalized, there exist trade-offs between a natural formalization and the formalization that would produce the most accurate informalization to match the original expressions in the book. These trade-offs arise when the formal representation lacks a direct correspondence to the original informal expression. Different approaches were tested to match existing formalization and align with the informal base. The results indicate that Informath is able to informalize Homotopy Type Theory, but the structure of the formalization may need to be structured around the desired result to get a matching informalization.

Keywords: Homotopy, type theory, Dedukti, Grammatical Framework, formalization, informalization, Informath.

Acknowledgements

Firstly, I would like to thank Aarne Ranta who has been wonderful to work with, and has acted as an excellent supervisor throughout the process.

Secondly, I would like to thank my examiner Ana Bove who has ensured that the project got finished on time and gave valuable and honest feedback throughout the project.

I would also like to thank Inari Listenmaa was hugely helpful with understanding the technical details behind GF.

I'm thankful for everyone I met at the TYPES 2026 conference who showed interest and gave feedback on my work.

Lastly, I would like to thank Vincent Harbander for helping with the proof reading of the thesis.

May Ohlsson, Gothenburg, 2026-07-07

Contents

List of Figures

List of Tables

1

Introduction

Translation between informal mathematics, written for human readers, and formal mathematics, encoded in proof assistants, is a time-consuming and error-prone task. Attempts to automate this process have been tried in the past with probabilistic approaches, but these do not guarantee correctness of the original formalized theory. The Informath project explores informalization, turning formal mathematics into informal mathematics [1]. Informath aims to enable full translation by also performing autoformalization, turning informal mathematics into formal mathematics, but is currently focused on informalization. The core idea is to translate different proof assistants into an intermediate universal representation. This representation is then given a complete grammar to map its functions to informal representations. This approach allows for an extendable grammar, that can enable informalization into multiple natural languages from multiple different proof assistants. This thesis investigates whether Informath can be extended to handle more advanced mathematics, using Homotopy Type Theory (HoTT) as a case study. HoTT is a new field with active development that explores many new ideas, while also building on type theory, which made it an excellent candidate to informalize. We identify the challenges that arise and discuss how they might inform the future development of Informath.

1.1 Problem description

This thesis has two main parts:

- Extend Informath by formalizing the definitions presented in Rijke's book *Introduction to Homotopy Type Theory* and create a grammar to enable informalization to a representation that matches the original expressions from the HoTT book [2]. The definitions should be type correct and be presented with a reproducible demo.

- Assess whether Informath is able to generate an accurate informalization of HoTT. This should be done by documenting the process and presenting the challenges that arise. Then we should conclude whether Informath is currently the appropriate tool to informalize advanced mathematics.

1.2 Related Work

Much effort is currently being put into formalizing mathematics within proof assistants. Active projects such as the *Archive of Formal Proofs* in Isabelle [3], *UniMath* in Agda [4], and *Mathlib* in Lean [5] are aim to formalize mathematics in their respective proof assistants.

The idea of informalizing mathematics is not new, with numerous recent projects using large language models (LLMs) for autoformalization and informalization.

In 2018 this approach was first tried by training LLMs to translate informal LaTeX-text of Mizar into formal Mizar code by Wang et al. [6]. Informalization was used to create formal-informal pairs for the training data.

It was later expanded on by leveraging the development of LLMs and existing formalized mathematics by Jiang et al. [7]. LLMs were used for the informalization of Archive of Formal Proofs and Mathlib, which gave promising results.

Rijke's book, *Introduction to Homotopy Type Theory*, has already been formalized in Agda [8]. Continued work on this has been done in the Unimath project to expand, modernize and standardize the initial formulation.

1.3 Motivation

There are numerous use cases of tools that are capable of informalizing and autoformalizing mathematics. Modern mathematics increasingly relies on proof-assistants, and formal proofs generated by LLMs [9]. Translating such formal mathematics into a form readable by those unfamiliar with coding or proof-assistants would increase its accessibility.

As noted earlier, Informath could act as a tool used to formalize mathematics for an LLM to more effectively comprehend and work with. This is done through a process of creating synthetic data.

A major limitation of prior works on informalization and autoformalization has been how hard it is to create formal-informal pairs for training data to train LLMs. According to Jiang et al. [7, p. 1]:

The most challenging part of autoformalization research is constructing such a parallel dataset in a natural and a formal language [...].

Informath can convert existing formally verified mathematical theories into such parallel datasets. It could also allow an LLM to use existing verification tools in one of the proof-assistant languages to mitigate the hallucination issue.

Another use case would be to act as an aid for visually impaired people to understand the technical content of the book, because each symbolic formula is required to have a fully verbal representation.

Finally, once something has been formalized, minimal effort is required to create a

full mapping to another language. This could be used to translate textbooks or to act as an aid to translate specific parts.

1.4 Autoformalization and Informalization

The gap between formal and informal mathematics can be bridged with autoformalization and informalization. Autoformalization is computer translation from informal to formal mathematics and is now more relevant than ever due to the development of LLMs. A system from Google DeepMind was used to solve the problems in the 2024 International Mathematical Olympiad [10]. It ended up scoring on par with the silver medalist level, showcasing the problem solving ability of this model. However, the process by which this was done reveals the importance of formalization. Instead of giving the LLM the problems, they were first translated into formal mathematical expressions in order for their model to be able to handle the problems. This creates the question whether autoformalization is possible, which means automating this translation. This could eliminate the human step in the process and make the model completely independent. There are two main approaches to autoformalization: a non-deterministic one using LLMs and a deterministic one using a formal grammar. This deterministic approach is limited by only being able to translate what is already in the grammar.

Informalization is the opposite of this, translating formalized mathematics into informal natural language. This has been the approach to generate the formal-informal pairs to train an LLM in earlier works.

1.5 Thesis outline

The rest of the thesis is divided into the following chapters:

- **Background:** Presents the structure and motivation behind Informath and how to use it. The theories that will be covered are dependent type theory and HoTT.
- **Method:** Explains how to extend Informath to cover a formalized theory, using HoTT as the example.
- **Results:** Presents the resulting informalization of HoTT.
- **Discussion:** Addresses the challenges encountered and assess whether Informath is the right tool for this type of task.

2

Background

This chapter presents the tools Informath is built around, Informath itself and a summary of the theory that is needed to understand the project.

2.1 Proof assistants

A proof assistant is a tool designed to help formulate and validate a proof. The first implementation that used computers to formally check mathematical proofs was the Automath project that started in 1967 [11].

Today, proof assistants are often based on type theory allowing for a direct equivalence between types and logical propositions. Relevant examples to this thesis include Agda [12], Lean [13] and Rocq [14].

2.2 Grammatical Framework

The language processing part of Informath is handled by Grammatical Framework (GF), a functional programming language designed for multilingual grammar applications [15].

When using GF for translation, one defines a shared grammar file that contains all functions needed to represent the given abstract syntax. A separate concrete syntax file is then created for each supported language, defining how the abstract syntax is rendered in that language. A simplified example inspired by the GF tutorial [16] is shown in Listings 2.1, 2.2 and 2.3.

```
Is : Topic -> Quality -> Phrase ;
HoTT : Topic ;
Very : Quality -> Quality ;
Interesting, Difficult, Fun : Quality ;
```

Listing 2.1: Defining the shared syntax for possible expressions.

```
Is topic quality =
{s = topic.s ++ "is" ++ quality.s} ;
HoTT = {s = "HoTT"} ;
Very quality =
{s = "very" ++ quality.s} ;
Interesting = {s = "interesting"} ;
Difficult = {s = "difficult"} ;
Fun = {s = "fun"} ;
```

Listing 2.2: English mapping of the syntax.

```
Is topic quality =
{s = topic.s ++ "är" ++ quality.s}
   ;
HoTT = {s = "HoTT"} ;
Very quality =
{s = "väldigt" ++ quality.s} ;
Interesting = {s = "intressant"} ;
Difficult = {s = "svårt"} ;
Fun = {s = "kul"} ;
```

Listing 2.3: Swedish mapping of the syntax.

We generate the tree for a sentence in this grammar in Figure 2.1.

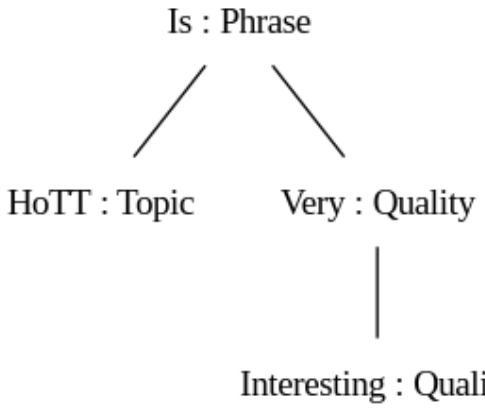


Figure 2.1: Tree for the sentence “HoTT is very interesting.”

Translation can be done by parsing a sentence in one language, and linearize it in another language. This approach for translation allows for a straightforward process of adding support for multiple languages [15].

This approach is what has enabled the creation of the GF Resource Grammar Library (RGL) [17]. It is the standard library for GF, consisting of a collection of parallel grammars for many languages constructed around one common abstract syntax.

2.3 Dedukti

Dedukti is a logical framework designed primarily to type check proofs generated by proof assistants [18]. It is not itself a proof assistant, but it is used as a type checker for the proof assistant $\lambda\Pi$ [19]. Proof assistants such as Agda, Lean and Rocq, have different strengths. For each of these, there exist translators of various stages of completion between them and Dedukti [20] [21] [22]. These other proof assistants have tools and tactics specifically designed to help the user write proofs, such as creating cases or filling in gaps in a proof, which is not the case in Dedukti. There is also ongoing work on translating from Dedukti to these proof assistants [1]. Dedukti

being built as a universal proof checker does allow for it act as an in-between step in translation. This universality can be used to translate formal mathematics to informal mathematics by only having to informalize Dedukti theory rather than the theory from all the other proof assistants. Additionally it can be used to translate between two different proof assistants.

2.3.1 λΠ-calculus

Dedukti is defined by a minimal syntax and is based on the λΠ-calculus modulo, which can be seen as normal λ-calculus extended with dependent types and a richer context. Types in λΠ-calculus are terms of the type $Type$. From this one can construct expressions of the form $A \to Type$ allowing for the construction of types that are dependent on a given input A. Another general type is $Kind$, which is the type for $Type$ itself and terms such as $A \to Type$. Dependent functions are constructed with the Π-type, and a type family B that depends on $x : A$ is written $\Pi x : A\ B$. The symbol Γ is used to represent a context, which is a list of variable declarations. With this we can write $\Gamma \vdash a : A$ to state "In context Γ, the term a has type A." The following is a summary of the typing rules that are used in λΠ-calculus from the original Dedukti paper [18]:

An empty context is well formed.

$$\frac{}{[\,] \text{ well-formed}}$$

A type or type family variable can be added to a context.

$$\frac{\Gamma \vdash A : Kind}{\Gamma \vdash x : A \text{ well-formed}}$$

An object variable can be added to a context.

$$\frac{\Gamma \vdash A : Type}{\Gamma \vdash x : A \text{ well-formed}}$$

$Type$ is a term of type $Kind$.

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash Type : Kind}$$

Variables can be extracted from well-formed contexts.

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A} x : A \in \Gamma$$

Product types can be constructed.

$$\frac{\Gamma \vdash A : Type/Kind \qquad \Gamma, x : A \vdash B : Type/Kind}{\Gamma \vdash \Pi x : A\, B : Type : Kind}$$

Functions are terms of product types.

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : Type/Kind \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\, t : \Pi x : A\, B}$$

Functions can be applied to terms.

$$\frac{\Gamma \vdash t : \Pi x : A\, B \qquad \Gamma \vdash t' : A}{\Gamma \vdash (t\, t') : (t'/x)B}$$

Terms of a type can be converted to congruent types and kinds.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Type/Kind \qquad \Gamma \vdash B : Type/Kind}{\Gamma \vdash t : B} A \equiv_\beta B$$

The $\lambda\Pi$-calculus modulo theory is an extension of $\lambda\Pi$-calculus where the global context can also include rewrite rules, which we will see in Section 2.3.2. Typing is preserved under reduction. It also allows for a global context, that contains variables and rewrite rules that can be used everywhere, and local contexts, that only contains variables that can be used within that context.

2.3.2 Example

Dedukti is a proof checker for $\lambda\Pi$-calculus modulo. We have seen how both $Type$ and $Kind$ are handled by $\lambda\Pi$-calculus, but in Dedukti, $Kind$ is used only internally. This means $Type$ is the only base constant exposed to the user. We introduce the syntax of Dedukti by looking at some simplified examples from the official guide [23] shown in Listing 2.4.

```
Nat : Type.

zero : Nat.
succ : Nat -> Nat.

def plus : Nat -> Nat -> Nat.
[ n ] plus zero n --> n
[ n, m ] plus (succ n) m --> succ (plus n m).
```

Listing 2.4: Definition of the natural numbers and addition in Dedukti.

Here we introduce the type Nat for the natural numbers, the zero constant and the successor function with their respective type. We also introduce our first function, plus, which demonstrates the use of rewrite rules. Variables inside the square brackets are in a local context and are the ones being pattern matched on.

A different notation can be used for definitions that only require one rewrite rule, making it possible to formalize in one line. The two definitions below are equivalent.

```
def one : Nat.
[] one --> succ zero.

def one : Nat := succ zero.
```

Listing 2.5: The two ways to define rewrite rules in Dedukti.

This approach only works when no pattern matching is required. One can do this for functions that are typically defined by pattern matching by explicitly giving the arguments through leading λ-abstraction.

We can illustrate dependent types in Dedukti, with the example of arrays indexed by their length. We introduce arrays as a family of types that are parameterized by their size. Arrays can be inductively defined in terms of the empty array, `nil`, and constructor `cons`, that adds an element to the head of the array. We can then define an `append` function that concatenates two arrays that are parameterized by their length, and a safe `head` function. All of these examples are shown in Listing 2.6.

```
array : Nat -> Type.
nil : array zero.
cons : (n : Nat) -> A -> array n -> array (succ n).

def append : (n : Nat) -> array n -> (m : Nat) -> array m ->
array (plus n m).
[ n, v ] append zero nil n v --> v
[ n, v1, m, v2 ] append (succ n) (cons n v1) m v2 -->
cons (plus n m) (append n v1 m v2).

def array_head : (n : Nat) -> array (succ n) -> A.
```

Listing 2.6: Showcase of arrays parameterized by their length.

In the example above, whenever n and m are declared, they are explicit identifiers, so that array can depend on the length. The `array_head` function in Listing 2.6 is only well defined for arrays with at least length one. The explicit length of each array lets the Dedukti type checker evaluate well formed expressions, such as

```
#EVAL append zero nil zero nil.
```

to `nil`, but

```
#EVAL append (succ zero) nil zero nil.
```

causes a type error.

Dedukti supports λ-expressions, where $\lambda x.fx$ would be expressed as `x => f x`.

This is a taste of the syntax and features of Dedukti. There are more features, but those are outside of the scope of what is required to understand the work.

2.4 Informath in depth

We must first distinguish between what Informath currently is and the project's goals. Informath is currently mainly used to informalize Dedukti files [1]. The informalization defaults to English, but can be extended to any other language supported by the grammar, which currently are Swedish, French and German.

Informath aims to be able to informalize theories written in Agda, Lean and Rocq. These proof assistants all have tools to enable translation into Dedukti. Rudimentary translation from Dedukti into them is possible but it is not in a state to be applied to this project.

The overarching structure of Informath is shown in Figure 2.2.

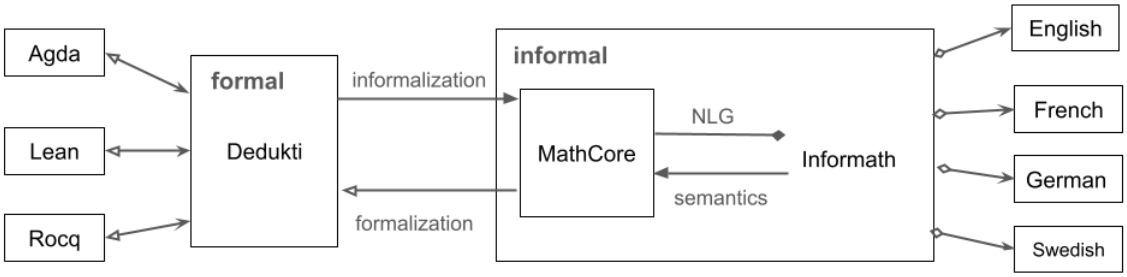


Figure 2.2: Structure behind Informath. Arrows indicate translation from one representation to another [1].

Informath itself is not a formal proof-assistant. Rather, it is a translation framework, which builds upon several components. The primary component is GF, which is used to construct the grammar for natural language [15]. Informath translates formal Dedukti theories into GF functions, allowing users to easily informalize existing formally verified mathematics.

Type correctness is handled by the Dedukti type checker for the formalized theory. Dedukti was chosen because of its similarity in structure to GF, minimal syntax and the existence of translations tools into it.

2.4.1 Structure of Informath

There is the upper layer of Informath that a user is expected to modify to use the program. Users are required to formalize their theory in Dedukti and associate it with a grammar. The symbol table, the mapping between Dedukti functions and informal expressions, contains base constants which are used by default, but can be extended by adding the definitions to a **.dkgf** file. Informath can informalize a valid Dedukti file even without an explicit grammar, but the results will then only be based on the base constants. The Dedukti function *plus* can be added by linking it as shown in Listing 2.7. In that mapping, X and Y are interpreted as arguments by Informath.

```
plus : "the sum of X and Y"
```

Listing 2.7: A simple link between a Dedukti function and an informal representation

Next is the grammar component. The base grammar currently consists of a collection of over 4,000 words, but this can be extended by a user if the words needed are not in the grammar. The process of adding words to the grammar can be partly automated by Universal Dependencies (UD) [24]. The grammar builds on top of MathCore, which is the core abstract-syntax for the project. Natural Language Generation (NLG) is the conversion to turn the grammar into informal expressions. Additional modules extend this foundation to cover a wider range of mathematics. Informath has extensive utility functions to simplify the process of expanding the grammar. To add a lexical item such as **zero** to the grammar, one would need to add it to the general grammar file and the language-specific one as shown in Listing 2.8.

```
# Example.gf
fun zero_Noun : Noun ;

# ExampleEng.gf
lin zero_Noun : mkNoun zero_N ;
```

Listing 2.8: Example of adding a noun to the GF grammar.

Finally there is the lower layer of GF functions and backend code. When a required GF function is not present in Informath, one could add such a function. For example, this could be if a function needs to take in more than 3 arguments, as the current version of Informath does not support such a function. This is also the layer where the LaTeX-document is generated. One could change the default included packages when generating the document, or making other changes that should be applied to each document in this manner.

2.4.2 The informalization process

Informath generates informalizations of formal expressions by enumerating all variations that are valid in the given grammar. This can produce a large number of variations, depending on the size of the grammar and the complexity of the expression. In order to get a reasonable result, all the variations are ranked based on a scoring function to decide on the best one. The main factors are:

- the length of the informalization, measured in tokens and characters
- the length and depth of the abstract syntax tree created by GF
- if an expression starts with a symbolic or verbal expression.

This is done to favor simple results, to prevent LaTeX-expression from starting with a formula and keep them in a single line. The number of variations grows exponentially as the expression size increases, while the running time grows even faster since it needs to generate all the variations and sort them to select the best candidate.

Informath is able to generate an informalization for any valid Dedukti file, even without a grammar. It would generate the following text for the example function *plus*:

Plus. Let x and y be elements of Nat. Then Nat. By cases:

- *for n, plus applied to zero and n is n.*
- *for n and m, plus applied to succ applied to n and m is succ applied to plus applied to n and m.*

We note that the rewrite rules are expressed as cases. The informalization can be improved by extending the grammar.

2.4.3 The .dkgf file format

Informath uses a **.dkgf** file to extend its grammar to Dedukti functions. Its basic function is to link a Dedukti function to a GF function which is shown below.

```
zero : zero_Name
succ : successor_Fun
plus_Oper2 : Oper2
```

Listing 2.9: Example of how one can extend the grammar.

It also allows for some more advanced functionality to make it easier to use. In practice, one would define the mapping in a simpler way as the following,

```
zero : "zero"
succ : "the successor of X"
plus : "the sum of X and Y"
```

Listing 2.10: Example of an easier way to link Dedukti functions to the grammar.

where X and Y are understood to be arguments. If a function has a verbal representation, i.e., a corresponding phrase, it is possible to extend this to a symbolic one by giving it a LaTeX-macro with the right number of arguments as an option. One would often rather create a specific LaTeX-command which is also easy to do using the `#MACRO` pragma. An example of this features is shown in Listing 2.11.

```
#MACRO \newcommand{\plus}[2]{#1 + #2}
plus : "the sum of X and Y" | \plus
```

Listing 2.11: Showcase of how arrays can be parameterized on their length.

which allows Informath to generate the following informalization:

Definition. Let x and y be natural numbers. Then $x + y$ is a natural number. By cases:

- *for n, $zero + n$ is n.*
- *for n and m, $succ(n) + m$ is the successor of $n + m$.*

The **.dkgf** format also supports dropping arguments from a function so it can be interpreted as a different arity function by GF. This comes in handy because Dedukti does not support implicit arguments so it is often desirable to drop these leading arguments. This is done with a #DROP function_name N pragma. This will drop the first N arguments from a function. Instead of mapping to a GF function, it is often easier to map directly to a sentence. This can be done as long as the words in the sentence are already in the grammar, and if the sentence has the correct structure. The way this is down is shown in Listing 2.12.

```
#DROP concat 4
#MACRO \newcommand{\concat}[2]{#1 \cdot #2}
concat : "concatenation." | "the concatenation of X and Y" | \concat

#DROP htpy 2
#MACRO \newcommand{\homotopy}[2]{#1 \sim #2}
htpy : "X is homotopic to Y" | \homotopy
```

Listing 2.12: Example of the rest of the features of the .dkgf file format.

Definitions in the HoTT book usually fall into one of two cases. Definitions with and without rewrite rules. Since the goal is to create an informalization that matches the one in the HoTT book, it can be the case that informalizing just the type declaration is enough to match the expressions in the HoTT book. In other cases the type declaration is too general, meaning the rewrite rules are needed. We can show both of these off with some examples.

In the first case, we have `refl`, of the type `(A : U) -> (a : El A) -> El (Id A a a).` which is enough to create an informalization. In the second case, we have `plus`, of the type `Nat -> Nat -> Nat`. This is too general since this could match many functions, such as `mult`.

2.5 Theory

While a full explanation of HoTT is outside of the scope of this thesis, the basic concepts needed to understand the terminology and main ideas are presented here. This section is divided into two parts, corresponding to the first two chapters of Rijke's book [2].

2.5.1 Martin-Löf Dependent Type Theory

The book devotes the first eight sections to presenting Martin-Löf Dependent Type Theory (MLDTT) [25]. Much of current mathematics is built on set theory, but this is not the only foundation for mathematics [26]. Type theory classifies all mathematical objects into types. This has proved especially useful in computer science where many proof assistants are based on type theory, such as Lean, Agda and Rocq. This is because you can encode logic into type theory, which we see in Table 2.1.

There are two primitive objects in type theory: types and elements. The difference between simply typed type theory and dependent type theory is the inclusion of

dependent types. In dependent type theory, both elements and types can be parameterized by elements of another type. By convention, capitalized letters will be used to signify a type, while lowercase letters will represent elements.

2.5.2 The representation of dependent type theory

Another core concept is the idea of type families. We denote x being of type A as $x : A$. We say that B is a type family if $B(x)$ is a type for all $x : A$. Here B is parameterized by x and the type of $B(x)$ will therefore depend on x. This gives rise to the dependent function type, which we will call the Π-type. We will use the notation $\Pi_{x:A}B(x)$ to represent dependent functions whose output type $B(x)$ depends on $x : A$.

Type families also give rise to dependent pairs. For a type family B over A, we have a dependent pair (a, b), where $a : A$ and $b : B(a)$. This dependent pair is usually denoted as $\Sigma_{x:A}B(x)$.

2.5.3 Notation

Other concepts from classic logic have an analogous representation in type theory. Disjunction is represented by the coproduct, $A+B$, and conjunction by the Cartesian product $A \times B$ for some types A and B. We also have types $\mathbf{1}$ and $\mathbf{0}$ that correspond to true and false. These are called the unit type and the empty type.

Formulating the components of mathematical logic into MLDTT is called the Curry-Howard correspondence [27]. This is shown in Table 2.1.

Logic	**Type Theory**
proposition	type
proposition true	unit type $\mathbf{1}$
proposition false	empty type $\mathbf{0}$
proof of P	term of type P
conjunction $P \wedge Q$	product type $P \times Q$
disjunction $P \vee Q$	coproduct type $P + Q$
implication $P \Rightarrow Q$	function type $P \to Q$
negation $\neg P$	function type $P \to \mathbf{0}$
existential quantification $\exists x : A.\, P(x)$	dependent pair type $\Sigma_{x:A}P(x)$
universal quantification $\forall x : A.\, P(x)$	dependent function type $\Pi_{x:A}P(x)$

Table 2.1: The Curry-Howard correspondence between logic and MLDTT.

2.5.4 Equality and universes

Equality between two elements $x = y$ usually means these have the same value and can be used interchangeably. In type theory, however, this is not the full story.

Concretely, for two elements, $x, y : A$, the expression $x = y$ denotes the *identity type*. With this interpretation, we can have multiple elements inhabiting such a type. For example, $p : x = y$ means that p is a witness that x and y are equal, we can then have $q : x = y$ which is a another witness of the equality. This allows for statements of equality of equalities, such as $p = q$.

The last concept to introduce is the idea of universes. They can be thought of as a type consisting of types. A universe is often denoted as $\mathcal{U}$. There are cases where we would want a universe to be an element of some other universe. A universe being an element of itself would lead to inconsistencies in the theory [2].

An assumption is made that there are always enough universes to ensure that every universe is an element of some larger universe. In other words, for any universe $\mathcal{U}_n$ there exists another universe $\mathcal{U}_{n+1}$ such that $\mathcal{U}_{n+1}$ contains $\mathcal{U}_n$ as an element, as well as all the types in $\mathcal{U}_n$. Universes allow for defining type families over inductive types by using their induction principle. Examples on how this is done can be found in Appendix A.

2.5.5 Homotopy Type Theory: The Univalent Foundations of Mathematics

Homotopy theory originates in algebraic topology, with the core idea being that maps can be related by homotopies. An intuitive way to explain a homotopy is to imagine you have two points on a space. Let there be two different paths between these two points. A homotopy between the paths is a continuous deformation of one path into the other. This gives rise to the study of topological structures up to continuous deformations.

HoTT is the interpretation of dependent type theory, where we treat types and elements as topological objects. This interpretation of type theory allows for many use cases that would otherwise be impossible with ordinary type theory. It is often impossible to prove that two functions are equal, even when one can show that they are pointwise equal. In fact, this pointwise equality is how homotopies are defined in HoTT [2]. A homotopy between f and g existing is denoted as,

$$f \sim g,$$

where $f, g : \Pi_{(x:A)} B(x)$. This core idea of treating types as topological spaces gives rise to the rest of the theory. Two spaces A and B are homotopy equivalent (denoted as $X \simeq Y$) if there exist maps $f : A \to B$ and $g : B \to A$ such that $f \circ g$ and $g \circ f$ are homotopic to the identity function. This idea leads to the most important principle in HoTT. The univalence axiom:

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B).$$

This states that identity type between two types is equivalent to the equivalence type between them.

Similarly to the Curry-Howard correspondence, we can translate some ideas from homotopy theory to concepts more familiar in type theory. See Table 2.2.

Type Theory	**Homotopy Theory**
types	spaces
dependent types	fibrations
elements	points
dependent pair types	total spaces
identity types	path fibrations

Table 2.2: Correspondence between type theory and homotopy theory from the HoTT book [2].

3

Methods

Extending Informath to support HoTT was not a simple, carefully planned process. The approach being novel, and the Informath project changing throughout the process has led to many different approaches being attempted. To document this effort and help others who may wish to follow a similar path, the following summary outlines the key stages.

3.1 Representation of HoTT in Dedukti

There are many ways a theory can be formalized. The most important decision is therefore picking a representation, since this will impact the entire formalization, and in turn the basis for informalization.

We have seen how Dedukti can be used to formalize theory and verify correctness through type checking. HoTT has been formalized in these proof assistants. Notably, Rijke's book *Introduction to Homotopy Type Theory* has almost been entirely formalized in Agda [8]. The first natural step was then to translate the Agda formulation of the book and create a grammar for the result. This approach turned out to be unsuccessful. The tool Agda2Dedukti [20] was only able to generate translations for the first couple of chapters. Even if the rest of the book could be translated this way, the translation does not lend itself to informalization without either parsing or a major rewrite. An example translation of an Agda function from the existing formalization is shown in Listing 3.1 [8].

```
-- Agda base --
ind- : {i : Level} {P :  → UU i} → P zero- → ((n :  ) → P n → P(succ-
   n)) → ((n :  ) → P n)
ind- p0 pS zero- = p0
ind- p0 pS (succ- n) = pS n (ind- p0 pS n)


-- Dedukti translation --
def {|!ind-|} : Agda.Term Agda.sortOmega (Agda.qLevel (i => Agda.set (
   univ.s i)) (i => Agda.prod (Agda.set (univ.s i)) (Agda.set i) (Agda
   .prod (Agda.set univ.0) (Agda.set (univ.s i)) {|!|} (_0 => (00
   preamble.UU i))) (P => (Agda.prod (Agda.set i) (Agda.set i) (P {|!
   __zero-|}) (_0 => (Agda.prod (Agda.set i) (Agda.set i) (Agda.prod (
   Agda.set univ.0) (Agda.set i) {|!|} (n => (Agda.prod (Agda.set i) (
   Agda.set i) (P n) (_0 => (P ({|!__succ-|} n)))))) (_0 => (Agda.
   prod (Agda.set univ.0) (Agda.set i) {|!|} (n => (P n))))))))))).
```

```
[i, P, x0, x] {|!ind -|} i P x0 x  {|!__zero -|} --> x0.
[i, P, x0, x, n] {|!ind -|} i P x0 x  ({|!__succ -|} n) -->
x n ({|!ind -|} i P x0 x n).
```

Listing 3.1: Original Agda definition and Dedukti translation.

This is caused by the translator encoding Agda's type theory on top of Dedukti. It could be possible to fix the original Agda formalization to make a full translation and then parse the results to get the relevant information, but this was not done in favor of having freedom in how to represent HoTT in Dedukti.

When possible, the representation was chosen to match the existing one in Agda. This was done to not have to do everything from scratch, but also so that future work could have an outline to follow.

3.1.1 Universes and dependent functions in Dedukti

A core idea in MLDTT is universes. The classical way of representing a universe in a logical framework such as Dedukti would be to encode the level of a universe with the natural numbers, which is how it is done in the Agda representation [28].

Because the existing Agda representation was used as a baseline, some decisions from that work have influenced the Dedukti representation. One of the main ones is the lack of explicit representation of the Π-type. Dependent functions are still used, but are represented as their direct type of the form $A \to U$ rather than the more typical representation from the book $\Pi_{(x:A)}B(x)$.

A close match to the Agda formalization of the homotopy function would get informalized by Informath as follows:

Definition. Let $l1, l2 \in$ *Level. Let* x *be an element of* $l2$ *applied to Level. Let* $A \in U_{l1}$*. Let* B *be a function from elements of* A *to universes of* $l2$*. Let* $x \in A$*. Let* f *and* g *be functions from elements of* x *to elements of* B *applied to* x*. Then we can say that* $f \sim g$*. By cases:*

- *for* $l1$*,* $l2$*,* A*,* B*,* x*,* f *and* g*,* $f \sim g$ *is* $f(x) = g(x)$*.*

This direct informalization can be hard to follow, mainly because all arguments have to be explicit in the informalization. It has several issues where x is defined twice since it is the default variable Informath assigns to informalized expressions. This is one way of informalizing the code, but greatly differs from the original HoTT book formulation [2, p. 104].

Another approach would be to reduce these explicit arguments in favor of readability by having one universe that contains all types. The following is a simplified informalization of the same definition where we only use one universe, have an explicit Π-type and use that we only have one rewrite rule.

Definition. Let $x \in U$*. Let* y *be a function from elements of* A *to elements of* U*. Let* f *and* g *be elements of* $\Pi_{x:A}B(x)$*. Then* $f \sim g$ *is an element of Type defined as the set of functions from elements* x *of* A *to elements of* $f(x) = g(x)$*.*

Explicitly showing the Π-type completely changes the representation from the Agda formalization, which forced many definitions to be formed around this choice to get a more accurate informalization.

The type system of Dedukti is strict about ensuring the correct types of arguments. This has the effect that it becomes impossible to define a function $A \to B$ where A and B are universes. That is because A is expected to be $Type$ but is a universe U. One can get around this by defining a function $U \to Type$. In the HoTT book, we call this the *universal type family* and is denoted as $\mathcal{T}(X)$ for some $X : U$ [2]. This is the type of the elements of X. We name this function $El : U \to Type$ in Dedukti to match existing literature [29].

3.2 Formalizing the theory in Dedukti

The majority of the work was done by going through the book, section by section, and translating the definitions in the HoTT book into Dedukti.

Despite the focus of this thesis being on informalizing HoTT itself, and not the type theory, later sections do build on what was introduced earlier on. Sections 3 to 6 covering the core ideas of MLDTT were fully formalized. This served both as a soft start in learning how to formalize a theory in Dedukti, but also ensured all the prerequisite functions for the HoTT sections were already formalized. Many definitions depend on concepts introduced earlier, meaning more than just definitions had to be formalized to ensure type correctness.

We have seen that the representation of universes and dependent types can change how the theory is formalized. Three main representations were extensively tested.

- Agda-aligned: Universes are defined by levels, type declarations are very general and no explicit dependent function types.
- Book-aligned: Types correspond directly to the function arguments in the book. There is only one universe type. Only type declaration is defined as it encodes everything that needs to be informalized.
- Simplified: Only one universe is assumed. Certain functions have simplified types, but are given a full type definition to encode information that will be informalized.

Each approach was attempted for the full translation. Approaches were either abandoned when they proved infeasible to build around, or when another approach produced a better match to the original HoTT book's representation.

3.3 Extending Informath

Once a theory has been formalized in Dedukti, the functions are given a verbal representation. It is often enough to add one verbal and one symbolic representation

for each function. The simplest way to test if the representation worked is to generate and inspect the informalization.

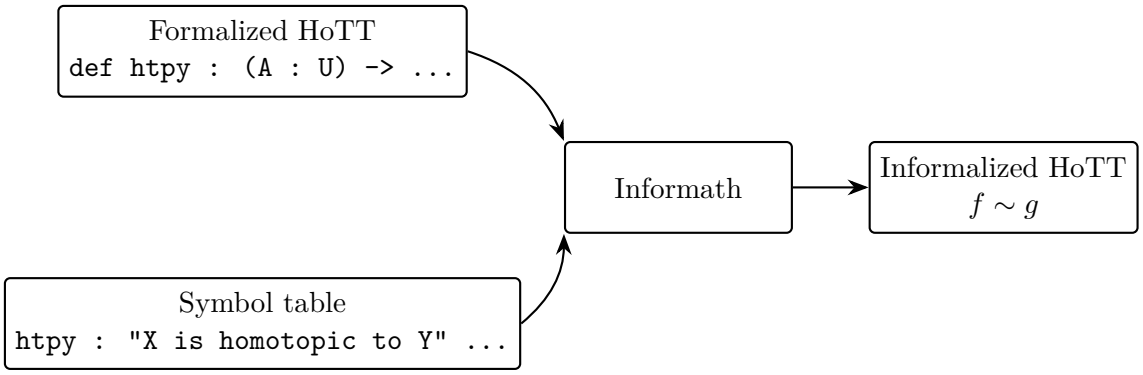


Figure 3.1: An overview of the informalization process.

It is possible to identify exact issues by generating the full JSON output containing all information Informath is working with [1]. This makes it possible to see exactly which representation is being chosen and which arguments are included. This proved not to be a practical approach for humans in most cases. It is possible to create a parser or compare the differences between two of these outputs after making a change, but even this was usually not practical.

As the size of the Dedukti theory and **.dkgf** file grows, so does the time to test the results. The most practical approach is to temporarily comment out unnecessary functions in the theory once they type check and have been tested if they are not needed.

3.4 Multiple language support

The process of extending informalization capabilities to another language is streamlined thanks to Informath being built around GF [1].

A full translation to different languages was not created, but a small demo was tested as a proof of concept. This is done by creating pairs of concrete syntax files in the targeted language and including them in *UserExtensions.gf* and its corresponding language-specific files (i.e, *UserExtensionsEng.gf* or *UserExtensionsSwe.gf*).

It is easy to identify words that are missing inside a langauge by attempting to informalize a theory into the language and looking at the result. An example of this for Swedish is shown below:

```
Definition. Låt $A$ vara ett element av [universe_Noun].
Låt $a$ och $b$ vara [element_Noun] av $A$.
Låt $x$ vara en [element_Noun] av [equality_Noun] typen mellan $a$ och
$b$. Då är [inverse_Noun] av $x$ en [element_Noun] av
[equality_Noun] typen mellan $b$ och $a$.
```

This makes it clear exactly which words need to be added to the grammar in order to get a full informalization.

4

Results

The definitions from section 9-15 in the HoTT book were formalized in Dedukti and linked with a **.dkgf** file to be informalized. The result is presented next to the original expression and can be reproduced by using the code presented in Appendix A.

All the translated definitions type-check in Dedukti and have been given informal representations. Of the three different approaches of formalization that were tested, only the last one was suitable for both formalizing the book and producing an informalization that matches the book's original text. Trying to match the Agda representation added too much information not present in the book representation, such as universe levels, while also leaving out other information, such as dependent function types. Trying to encode all relevant information in the types of functions produced the best informalization quality, but made the Dedukti code so complex that it was impractical to maintain past a point.

4.1 Informalization

A collection of informalized definitions and their original representation from the HoTT book are shown below to demonstrate the quality of the informalization. Since Informath is only capable of generating informalizations that are type correct, it is meaningless to compare its correctness to that of earlier probabilistic approaches.

HoTT book definitions [2]

Definition 9.1.2 Let $f, g : \Pi_{x:A} B(x)$ be two dependent functions. The type of homotopies from from f to g is defined as the type of pointwise identifications, i.e., we define

$$f \sim g := \Pi_{x:A} f(x) = g(x).$$

Informalization

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f and g be elements of $\Pi_{x:A} B(x)$. Then $f \sim g$ is an element of $Type$ defined as the set of functions from elements x of A to elements of $f(x) = g(x)$.

In this example, we see how Informath must introduce A and B, while these are assumed to be known in the book. The functions f and g are defined in the same way as the book. The definition has the same form but instead of `:=`, Informath

uses the verbal expression “is an element of”.

Definition 9.1.5 For any type family B over A we define the operations on homotopies

refl-htpy: $\Pi_{(f:\Pi_{x:A}\, B(x))}\, f \sim f$

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f be an element of $\Pi_{x:A}B(x)$. Then refl-htpy(f) is an element of $f \sim f$.

inv-htpy:
$\Pi_{(f,g:\Pi_{x:A}\, B(x))}\, (f \sim g) \to (g \sim f)$

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f and g be elements of $\Pi_{x:A}B(x)$. Assume that $f \sim g$. Then inv-htpy(H) is an element of $g \sim f$.

concat-htpy: $\Pi_{(f,g,h:\Pi_{x:A}\, B(x))}$
$(f \sim g) \to ((g \sim h) \to (f \sim f))$

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f, g and h be elements of $\Pi_{x:A}B(x)$. Let H be an element of $f \sim g$. Let K be an element of $g \sim h$. Then concat-htpy(H, K) is an element of $f \sim h$.

In these examples we see how once a definition has been given an informal representation, it can be used in other definitions. Everything that can have a symbolic representation is given one but Informath offers no simple way to informalize $\to$ without major changes to the formalization.

Definition 9.1.7
We define the following whiskering operations on homotopies:

(i) Supposed $H : f \sim g$ for two functions $f, g : A \to B$, and let $h : B \to C$. We define

$h \cdot H := \lambda x.\mathrm{ap}_h(H(x)) : h \circ f \sim h \circ g.$

Definition. Let A, B and C be elements of the universe. Let f and g be maps from A to B. Let h be a map from B to C. Let H be a homotopy between f and g. Then $h \cdot H$ is an element of $(h \circ f) \sim (h \circ g)$.

(ii) Supposed $f : A \to B$, and $H : g \sim h$ for two functions $g, h : B \to C$. We define

$H \cdot f := \lambda x.H(f(x)) : g \circ f \sim h \circ f.$

Definition. Let A, B and C be elements of the universe. Let g and h be maps from B to C. Let H be a homotopy between g and h. Let f be a map from A to B. Then $H \cdot f$ is an element of $(g \circ f) \sim (h \circ f)$.

The whiskering operations match the book closely but $\lambda x.H(f(x))$ and $\lambda x.\mathrm{ap}_h(H(x))$

can not be informalized since they are not part of the formalization.

Definition 10.1.1 We say that a type A is contractible if it comes equipped with an element of type

$$\text{is-contr}(A) := \Sigma_{c:A}\Pi_{x:A}c = x.$$

Definition. Let A be an element of the universe. Then is-contr(A) is an element of the universe defined as $\Sigma_{c:A}(\Pi_{x:A}(c = x))$.

This example shows it is possible to chain Π-types and Σ-types together.

Definition 11.5.2 Let A and B be types. We define

$$\text{Eq-copr}_{A,B} : (A+B) \to (A+B) \to \mathcal{U}$$

by double induction on the coproduct, postulating

$$\text{Eq-copr}_{A,B}(\text{inl}(x), \text{inl}(x')) := (x = x')$$

$$\text{Eq-copr}_{A,B}(\text{inl}(x), \text{inr}(y')) := \emptyset$$

$$\text{Eq-copr}_{A,B}(\text{inr}(y), \text{inl}(x')) := \emptyset$$

$$\text{Eq-copr}_{A,B}(\text{inr}(y), \text{inr}(y')) := (y = y')$$

Definition. Let A and B be elements of the universe. Let x and y be elements of $A + B$. Then Eq-copr$_{A,B}xy$ is an element of the universe. By cases:

- for A, B, x and x', Eq-copr$_{A,B}$inl(x)inl(x') is $x = x'$.
- for A, B, x and y', Eq-copr$_{A,B}$inl(x)inr(y') is $\mathbf{0}$.
- for A, B, y and x', Eq-copr$_{A,B}$inr(y)inl(x') is $\mathbf{0}$.
- for A, B, y and y', Eq-copr$_{A,B}$inr(y)inr(y') is $y = y'$.

In the above example, we see how multiple rewrite rules are handled. Informath does this by case distinction, which is a direct one to one mapping to the original expression, except that all arguments are made explicit in each case. Informath allows for three arguments at most, but by overloading `Eq-copr` we are able to give it four arguments to match the book representation. This has the drawback of being unable to provide a proper symbolic representation so the x and y arguments are appended to the end in Eq-copr$_{A,B}xy$.

Definition 12.1.1 A type A is said to be a proposition if its identity types are contractible, i.e., if it comes equipped with a term of type

$$\text{is-prop(A)} := \Pi_{(x,y:A)}\text{is-contr}(x = y)$$

Definition. Let A be an element of the universe. Then is-prop(A) is an element of the universe defined as $\Pi_{x:A}(\Pi_{y:A}\text{is-contr}((x = y)))$.

The `is-prop` example shows how $\Pi_{x,y:A}$ can be informalized by chaining together $\Pi_{x:A}$ and $\Pi_{y:A}$.

Definition 13.3.1 Given two propositions P and Q, we define their disjunction

$$P \vee Q := ||P + Q||$$

Definition. Let P and Q be elements of the universe. Then $P \vee Q$ is an element of the universe defined as $||(P + Q)||$.

Definition 13.3.1 Given a family P of propositions over a type A, we define the existential quantification $$\exists_{(x:A)} P(x) := \|\Sigma_{x:A} P(x)\|$$	Definition. Let A be an element of the universe. Let P be a function from elements of A to elements of the universe. Then $\exists_{x:A} P(x)$ is an element of the universe defined as $\|(\Sigma_{x:A} P(x))\|$.
Definition 15.1.4 For any map $f : A \to X$ we define the image of f to be the type $$\text{im}(f) := \Sigma_{x:X} \|\text{fib}_f(x)\|$$	Definition. Let A and X be elements of the universe. Let f be a function from elements of A to elements of X. Then $\text{im}(f)$ is an element of the universe defined as $\Sigma_{x:X} \|(\text{fib}_f(x))\|$.
Definition 15.2.1 A map $f : A \to B$ is said to be surjective if there is an element of type $$\text{is-surj}(f) := \Pi_{b:B} \|\text{fib}_f(b)\|$$	Definition. Let A and B be elements of the universe. Let f be a function from elements of A to elements of B. Then is-surj(f) is an element of the universe defined as $\Pi_{y:B} \|(\text{fib}_f(y))\|$.

.

In the last few examples, we see how propositional truncation can be used in combination with the expressions we have already seen.

The common trend for every informal expression is that each argument has to be introduced within the definition each time, while this is rarely the case in the book. The resulting expression was often a close match, but would sometimes lack a full symbolic representation.

Support for informalization to Swedish and French was added for a couple of the definitions. Below are fully verbal representations of the inverse homotopy.

Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f and g be maps from A to B. Assume that f and g form a homotopy. Then g is homotopic to f.

Soit A un élément de l'univers. Soit B une fonction des éléments de A à des éléments de l'univers. Soient f et g des applications de A à B. Supposons que f est homotopique à g. Alors g est homotopique à f

Låt A vara ett element av universumet. Låt B vara en funktion från element av A till element av universumet. Låt f och g vara avbildningar från A till B. Anta att f är homotopiskt till g. Då finns det en homotopi mellan g och f.

4.2 Informath as a tool to informalize HoTT

The secondary goal of the project was to assess if Informath is the right tool to Informalize HoTT. While all of the theory can be formalized in Dedukti, doing so does not always lend itself to informalization in a natural way.

Functions that can take an arbitrary number of arguments lack proper representation in the current grammar in Informath. This was seen in how Π-types could only take

two arguments in the Dedukti representation. It is meant to represent a dependent function type from A to B, but in informal mathematics, a Π-type can take any number of arguments for the expression of the dependent function, since the function can depend on any number of arguments. The way this was worked around in the informalization was to chain Π-types, which means $\Pi_{x,y:A}$ gets informalized as $\Pi_{x:A}(\Pi_{y:A})$.

λ-functions lack a clear informal translation, and Dedukti does not support optional arguments. Together, these limitations restrict how functions can be defined.

Informath works to express basic operations and can informalize into symbolic results that match the ones in the book, but struggles for larger definitions composed of multiple expressions due to having to generate and rank the informalizations. The informalization often becomes quite large because all leading arguments have to be explicit.

In the showcase above we saw that some rewrite rules get informalized as:

By cases:

- for ....

which is required to work for a definition with multiple rules, but is also the natural way to write rewrite rules in Dedukti.

Multiple definitions required major rewriting to improve the informalization, due to limitations in how Informath treats Dedukti. The informalization could be improved by adapting the Dedukti representation by encoding more information in the types.

5

Discussion

We covered as much of the book as possible within the given time. Throughout this process, many challenges arose.

5.1 Reflection

Was the approach chosen in this report the best one? Probably not. Someone with more knowledge about the Dedukti and HoTT could have done a different formalization of the theory, which could have lead to a better result.

Because the HoTT book has already been formalized in Agda, more effort could have been invested into making the translation tool work. While the informalization would be limited to the original Agda formulation, it could have been a meaningful contribution to create an automatic translation pipeline.

The benefit of working with Dedukti is reliant on the translation tools from the proof assistants. These translation tools are still limited and are not always up to date. Without a full translation, the drawbacks of using Dedukti start to outweigh the benefits. Dedukti is rarely used to formalize theory, instead it acts as the backend type checker to *Lambdapi*, a proof assistant based on the same $\lambda\Pi$-calculus modulo theory with Dedukti compatibility [19].

HoTT being based on type theory did lend itself to be formalized in a logical framework such as Dedukti, although this can partly be attributed to the existing formalization in Agda being natural to translate. It is possible that if another area had been picked for the case study, that the formalization would have been more problematic.

The choice of representation mattered. There are however drawbacks. When formalizing a statement in a proof assistant, the main concern is a correct formalization. However, when enabling informalization of a definition, one must take into account all the information that should be explicit. This mean that to informalize already formalized theory, one would either have to settle for an informalization that may be flawed, or have to redo the formalization to be informalized into a more elegant expression.

Informath is still in active development. The approaches presented in this thesis have been adapted around what is currently possible. Many of the limitations that

appeared during the initial process have already been fixed. Some larger limitations around the Informath project are still present, which is shown in the next section.

5.2 Limitations

A lot of potential use cases rely on mathematics that have already been formalized in a proof assistant. This is an issue since the translation between these and Dedukti is limited at best. If this could change in the future, it would open up many opportunities to informalize huge libraries of already formalized mathematics.

A problem is even if these were to be translated, their original formulation in their respective proof assistant may rely on features that would not translate well or that do not exist in other proof assistants. We have seen this in how the Agda translation would encode its type system on top of Dedukti, meaning the informalization would have even more explicit information. It could be the case that having an intermediate representation like Dedukti has more drawbacks than benefits.

If Dedukti was more widely used or was able to translate to other proof assistant without causing definitions to become as complex, then the project could have a lot of potential. Even if full translation between Dedukti and the other proof assistants existed, there are still fundamental limitations because of the way proofs are structured in the different proof assistants. For example, the way to formalize a theory in Agda could greatly differ from how the same theory would be formalized in Rocq, and this will likely never be fully captured by a translation using Dedukti as the intermediate step.

Because only parts of the book were formalized, and later informalized, it makes it impossible to know if this approach would work for the rest of the book. From the chapters that were translated, it became clear that any form of representation came with trade-offs.

5.3 Problematic informalization

There were many terms that could not be informalized without major trade-offs.

5.3.1 Higher order functions

A basic example of the type of function that is hard to informalize is any function that has a function type, $A \to B$, as its codomain. An example of this is shown below.

```
U : Type.
El : U -> Type.

test : (A : U) -> (B : U) -> (C : U) -> (f : (El A -> El B)) ->
(g : (El B -> El C)) -> (El A -> El C).
```

Listing 5.1: Problematic function.

Here, a function from elements of A to elements of C would be the desired output. The parentheses for the last term do not help, instead $El\,A$ and $El\,C$ gets interpreted as two separate terms. This could be fixed by introducing a function such as $MapAB$ that encodes the correct information. This has the advantage of making it possible to change the verbal representation of mapping and giving it a symbolic representation. Instead of writing out "a function from A to B" we could have "$A \to B$".

5.3.2 Rewrite rules

Any function with rewrite rules creates a dilemma. We can take the function `htpy` as an example.

The natural way of doing this is the following,

```
def htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
(g : El (Pi A B)) -> Type.
[A, B, f, g] htpy A B f g -->
(x : El A) -> El (Id (B x) (f x) (g x)).
```

Listing 5.2: Definition of homotopy that is written in the usual Dedukti style.

which produces the following informalization:

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f and g be elements of $\Pi_{x:A}B(x)$. Then we can say that $f \sim g$. By cases:

- *for A, B, f and g, $f \sim g$ is the set of functions from elements x of A to elements of $f(x) = g(x)$.*

We note that only one rewrite rule is used, so we could rewrite this into a single line in Dedukti. This works well for simple functions, but in this case we must do a slight workaround,

```
def htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
(g : El (Pi A B)) -> Type :=
A => B => f => g => (x : El A) -> El (Id (B x) (f x) (g x)).
```

Listing 5.3: Definition of homotopy that uses the single line rewrite rule.

where λ-functions have been chained together. This is rarely the way someone would write Dedukti but it does result in a simpler informalization:

Definition. Let A be an element of the universe. Let B be a function from elements of A to elements of the universe. Let f and g be elements of $\Pi_{x:A}B(x)$. Then f is homotopic to g, if for all elements x of A, there is an element of $f(x) = g(x)$.

Both formalizations type-check but the second one has a more elegant informalization. This creates a trade-off between a natural formalization and a simpler

informalization. The most elegant solution would be for Informath to simply convert the first representation into the other one internally. Thus, we can get a clean informalization without creating unreadable code.

5.3.3 λ-functions

Another issue is λ-functions in general. They cannot be customized within the grammar, since they are part of the Dedukti code syntax, not a definable function. In practice λ-functions are mainly used inside of Π-types and Σ-types or in function definitions. In the former case, it has the effect of adding an argument to the dependent types. Currently Informath does not support macros with optional arguments, but we will showcase how optional arguments could be added in Section 5.4. In the latter case, we had to settle for the existing formalization. For example, function composition, $(f \cdot g)(x)$, which is defined through λ-functions as $g \Rightarrow f \Rightarrow x \Rightarrow g\,(f\,x)$ gets informalized as "the function that maps g, f, and x to g applied to f applied to x."

5.4 LaTeX-macros

Some definitions were shown together with a commutative diagram, which we also attempted to informalize. There was no problem formalizing all the information, but the issue was rather extending Informath to handle something that takes more than three arguments, and to create LaTeX-macros that can create a commutative diagram. While it was shown to be possible by editing the backend code of Informath to add a specified multi-line macro at document creation, it is currently not possible to add through the **.dkgf** file like normal macros. A similar issue came up when trying to create LaTeX-macros that take in optional arguments. The way this can be done is shown below.

The LaTeX-package *xparse* allows for case distinction, which allows for optional arguments. In this project, the main use cases for optional arguments were the Σ-type and Π-type. These are typically coded to only take 2 arguments, but are informalized to show three. The variable x in $\Pi_{x:A}B$ can thus be hard coded in cases where it is a free variable, but can be replaced with another variable such as y when needed. An example of this would be the following macro:

```
\text{\NewDocumentCommand{\Pitype}{m g g}
{\IfNoValueTF{#3}{\Pi_{x:#1}#2(x)}{\Pi_{#2:#1}#3}}}
```

Here we create a command, with two different cases depending on whether a third argument is given. When creating a macro for a commutative diagram, the TeX-compiler has other limitations. Although there are packages that allow a user to create commutative diagrams, there was issues when using it inside a macro.

A LaTeX-document is separated into different parts. In the initial part, a user can include packages, create macros etc, a issue is that the TeX-compiler uses the ampersand symbol differently in that part of the document. A way to circumvent this is shown below.

```
\usepackage{tikz-cd}

\newcommand{\sq}[8]{%
\begingroup%
\begin{tikzcd}[ampersand replacement=\&]%
#1 \arrow[r, "#6"] \arrow[d, swap, "#5"] \&%
#2 \arrow[d,  "#7"] \\%
#3 \arrow[r, swap, "#8"] \&
#4%
\end{tikzcd}%
\endgroup%
}

% You can now call on the function like this
\sq{A}{A'}{B}{B'}{f}{g}{f'}{h}
```

This produces the following.

$$\begin{array}{ccc} A & \xrightarrow{g} & A' \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\ B & \xrightarrow[h]{} & B' \end{array}$$

This type of macro could be generalized for more types of commutative diagrams, but doing so now requires a lot of editing of the backend code of Informath. Future changes in Informath could make such additions easier without changing the backend code.

5.5 Selection bias in the results

Because Informath uses a scoring function, as we saw in Section 2.4.2, based on specified factors when evaluating which informalized result is the "best" one, there will be a subjective element baked into the design of the scoring function. The scoring function is a best guess, meaning the informalized expressions may not be the best match to the HoTT book. The results in Section 4.1 are therefore not representative of what one would expect the output to be, rather a collection of the best outputs from all generated variations, subjectively judged by the author. The alternative would be to change the scoring function to get all the best informalization, but that could be a difficult task, where making improving the ranking of one good expression could harm the ranking of another. This is a limitation in judging the results.

5.6 Ethical concerns

Informath is a tool that can generate thousands of variations of a sentence from a single formalized function. This could be used to create synthetic data for purposes such as training LLMs for autoformalization and informalization. Improving the capabilities of LLMs could lead to job losses for the researchers and developers working in this field. The domain of informalization is already highly specialized, and tools such as Informath are more likely to act as an aid or as a point of comparison with probabilistic models than to cause job loss. The approach of using Informath to generate synthetic data is preferable to using real user data as it avoids the ethical question of ownership of the original data.

5.7 Use of AI

During the project, the use of AI was kept at a minimum. The times where it came of use was to understand error messages generated by Dedukti and to create the LaTeX-macros for the commutative diagram.

5.8 Future work

The natural extension to this work would be to formalize the rest of the book in Dedukti and extend the grammar.

The focus in this project was to formalize and informalize the definitions in the HoTT book, but the book has a nearly complete formalization in Agda already, including everything this report did not have time to cover. Another area that could be explored is enabling Informath to work directly with other proof assistants rather than using Dedukti as an intermediate step. It is possible that this is more feasible than achieving a complete translation between Dedukti and the other proof assistants.

Migrating the project to *Lambdapi* could be of interest as it has the same advantages as Dedukti, but is still under active development and is built to be a proper proof assistant [19].

Informath is far from a finished product. The natural approach would be to improve Informath's core capabilities rather than working around its current limitations. A better user experience with clearer error messages and a streamlined translation process would speed up future work with the tool. It is currently time consuming to identify issues at all levels of the process. Creating proper documentation and a user guide would make the project more user friendly for the future.

5.9 Conclusion

The initial goal was to translate all the definitions in the HoTT book to Dedukti and then informalize them. While the entire book was not covered, a large portion

was. We can confidently state that the mathematics in the book can be formalized in Dedukti, however, it is unlikely to be the best tool for such a task. Dedukti, being a minimal logical framework, can formalize the definitions in the HoTT book, but such a representation does not lend itself to informalization in many cases. Informalization is often difficult. It is possible that using something other than Dedukti would have led to other limitations.

Informath has been shown to be able to informalize the mathematics presented in the book. Doing so had trade-offs depending on the formalization chosen. A simple natural formalization does not always match how one would express it informally. Conversely, to obtain an informalization that matches the book, the formalization tended to become more complex.

Informath is not a finished product, but we have explored how it currently can be used. If one wants to informalize existing theory, then Informath is a great tool. If one wants to create a formalization that produces a certain informalization, then Informath is also a great tool. If one wants to take an existing formalization, and turn it into a specific informalization, then one must be ready for trade-offs that may arise.

Bibliography

[1] A. Ranta, *Informath: Informalization and autoformalization of formal mathematics*, Accessed: 25 November 2025, 2025. [Online]. Available: https://github.com/GrammaticalFramework/informath.

[2] E. Rijke, *Introduction to homotopy type theory*, 2022. arXiv: 2212.11082 [math.LO].

[3] "The Archive of Formal Proofs," Accessed: May 28, 2026. [Online]. Available: https://isa-afp.org.

[4] E. Rijke, E. Stenholm, J. Prieto-Cubides, F. Bakke, et al., *The agda-unimath library*. [Online]. Available: https://github.com/UniMath/agda-unimath/.

[5] The mathlib Community, "The Lean Mathematical Library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020, New Orleans, LA, USA: ACM, Jan. 2020. DOI: 10.1145/3372885.3373824. [Online]. Available: https://doi.org/10.1145/3372885.3373824.

[6] Q. Wang, C. Kaliszyk, and J. Urban, "First experiments with neural translation of informal to formal mathematics," in *Intelligent Computer Mathematics*, F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, Eds., Cham: Springer International Publishing, 2018, pp. 255–270, ISBN: 978-3-319-96812-4.

[7] A. Q. Jiang, W. Li, and M. Jamnik, "Multi-language diversity benefits autoformalization," in *Advances in Neural Information Processing Systems*, A. Globerson et al., Eds., vol. 37, Curran Associates, Inc., 2024, pp. 83 600–83 626. DOI: 10.52202/079017-2659. [Online]. Available: proceedings.neurips.cc/paper_files/paper/2024/file/984de836e696ba653bbfbbbfce31d3bc-Paper-Conference.pdf.

[8] Rijke, Egbert, *Agda: Agda formalisation of the Introduction to Homotopy Type Theory*, https://github.com/HoTT-Intro/Agda, 2026.

[9] P. Song, K. Yang, and A. Anandkumar, "Lean copilot: Large language models as copilots for theorem proving in lean," *arXiv preprint arXiv:2404.12534*, 2024. DOI: 10.48550/arXiv.2404.12534. [Online]. Available: https://arxiv.org/abs/2404.12534.

[10] D. Castelvecchi, "Deepmind and OpenAI models solve maths problems at level of top students," *Nature*, vol. 644, Jul. 2025. DOI: 10.1038/d41586-025-02343-x. [Online]. Available: https://doi.org/10.1038/d41586-025-02343-x.

[11] H. Geuvers and R. Nederpelt, "N.g. de bruijn's contribution to the formalization of mathematics," *Indagationes Mathematicae*, vol. 24, no. 4, pp. 1034–

1049, 2013, In memory of N.G. (Dick) de Bruijn (1918–2012), ISSN: 0019-3577. DOI: https://doi.org/10.1016/j.indag.2013.09.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019357713000700.

[12] U. Norell, "Towards a practical programming language based on dependent type theory," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:118357515.

[13] L. d. Moura and S. Ullrich, "The lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds., Cham: Springer International Publishing, 2021, pp. 625–635, ISBN: 978-3-030-79876-5.

[14] The Rocq Development Team, *The rocq prover*, 2026. DOI: https://doi.org/10.5281/zenodo.15149628. [Online]. Available: https://rocq-prover.org.

[15] A. Ranta, *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications, 2011, ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

[16] A. Ranta, *Grammatical framework tutorial*, https://grammaticalframework.org/doc/tutorial/gf-tutorial.html, Accessed: 2026-06-01, 2010.

[17] A. Ranta, "The gf resource grammar library," *Linguistic Issues in Language Technology*, vol. 2, Dec. 2009. DOI: 10.33011/lilt.v2i.1205. [Online]. Available: https://journals.colorado.edu/index.php/lilt/article/view/1205.

[18] A. Assaf et al., "Dedukti: A logical framework based on the λΠ-calculus modulo theory," *arXiv preprint arXiv:2311.07185*, 2023.

[19] Deducteam, *Lambdapi, a proof assistant based on the -calculus modulo rewriting*, https://github.com/Deducteam/lambdapi, Accessed: 2026-06-01.

[20] Deducteam, *Agda2dedukti : A translator from agda to dedukti and lambdapi*, https://github.com/Deducteam/Agda2Dedukti, Accessed: 2026-06-01.

[21] Deducteam, *Lean2dk: Wip translation from lean to dedukti*, https://github.com/Deducteam/lean2dk, GitHub repository. Accessed: 2026-07-07, 2023.

[22] M. Boespflug and G. Burel, "Coqine: Translating the calculus of inductive constructions into the λΠ-calculus modulo," in *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, ser. CEUR Workshop Proceedings, vol. 878, Manchester, UK, Jun. 2012, pp. 44–50. [Online]. Available: https://ceur-ws.org/Vol-878/paper3.pdf.

[23] Deducteam, *Dedukti tutorial*, https://github.com/Deducteam/Dedukti/blob/master/TUTORIAL.md, Accessed: 2026-06-01.

[24] M.-C. de Marneffe, C. D. Manning, J. Nivre, and D. Zeman, "Universal dependencies," *Computational Linguistics*, vol. 47, no. 2, pp. 255–308, Jul. 2021, ISSN: 0891-2017. DOI: 10.1162/coli_a_00402. eprint: https://direct.mit.edu/coli/article-pdf/47/2/255/1938138/coli_a_00402.pdf. [Online]. Available: https://doi.org/10.1162/coli_a_00402.

[25] P. Martin-Löf, *Intuitionistic Type Theory*. Napoli: Bibliopolis, 1984.

[26] A. R. Mahmoud, "Set theory and its foundational role in modern mathematics," *ResearchGate preprint*, 2025. DOI: doi.org/10.13140/RG.2.2.27374.32323.

[27] W. A. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, H. Curry, H. B., S. J. Roger, and P. Jonathan, Eds., Academic Press, 1980.

[28] G. Brunerie et al. "Homotopy type theory in Agda. "[Online]. Available: `https://github.com/HoTT/HoTT-Agda`.

[29] T. Felicissimo, "Representing Agda and coinduction in the $\lambda\Pi$-calculus modulo rewriting," M.S. thesis, LSV, ENS Paris Saclay, Université Paris-Saclay, Sep. 2021. [Online]. Available: `https://inria.hal.science/hal-03343699`.

A

Appendix 1

The code to reproduce the main result from the thesis is collected in this chapter.

A.1 Dedukti formalization

The following is the theory formalized into Dedukti. This formalization has the trade-offs mentioned in the report.

```
(; Required functions and definitions ;)

U   : Type.
def El  : U -> Type.

def Pi : (A : U) -> (f : (El A -> U)) -> U.
[A, B] El (Pi A B) --> (x : El A) -> El (B x).

Sigma : (A : U) -> (El A -> U) -> U.
pair  : (A : U) -> (B : (El A -> U)) -> (a : El A) -> El (B a) ->
    El (Sigma A B).

def pr1   : (A : U) -> (B : (El A -> U)) -> El (Sigma A B) -> El A.
[A, B, a, b] pr1 A B (pair A B a b) --> a.

def pr2   : (A : U) -> (B : (El A -> U)) -> (s : El (Sigma A B)) ->
El (B (pr1 A B s)).
[A, B, a, b] pr2 A B (pair A B a b) --> b.

def Prod : U -> U -> U.
[A, B] El (Prod A B) --> El (Sigma A (_ => B)).

def pair_prod : (A : U) -> (B : U) -> El A -> El B -> El (Prod A B).
[A, B, a, b] pair_prod A B a b --> pair A (_ => B) a b.

Id     : (A : U) -> El A -> El A -> U.
def refl   : (A : U) -> (a : El A) -> El (Id A a a).
def inv    : (A : U) -> (a : El A) -> (b : El A) -> El (Id A a b) ->
El (Id A b a).
def concat : (A : U) -> (a : El A) -> (b : El A) -> (c : El A) ->
El (Id A a b) -> El (Id A b c) -> El (Id A a c).

ap : (A : U) -> (B : U) -> (f : (El A -> El B)) -> (a : El A) ->
(b : El A) -> El (Id A a b) -> El (Id B (f a) (f b)).
```

```
tr : (A : U) -> (a : El A) -> (b : El A) -> El (Id A a b) ->
(B : (El A -> U)) -> El (B a) -> El (B b).

def id_fun : (A : U) -> El A -> El A.
[A] id_fun A --> x => x.

def fun_comp : (A : U) -> (B : U) -> (C : U) -> (El B -> El C) ->
(El A -> El B) -> El A -> El C.
[A, B, C] fun_comp A B C --> g => f => x => g (f x).

MapNondep : U -> U -> Type.
htpy_nondep : (A : U) -> (B : U) -> MapNondep A B -> MapNondep A B
-> Type.
comp_fun : (A : U) -> (B : U) -> (C : U) -> (MapNondep B C) ->
(MapNondep A B) -> (MapNondep A C).

(; Chapter 9 ;)

(; Definition 9.1.2 ;)

def htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
(g : El (Pi A B)) -> Type.
[A, B, f, g] htpy A B f g --> (x : El A) -> El (Id (B x) (f x)
(g x)).

(; Definition 9.1.5 I ;)
def refl_htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
   htpy A B f f.
[A, B, f] refl_htpy A B f  --> x => refl (B x) (f x).

(; Definition 9.1.5 II ;)
def inv_htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
(g : El (Pi A B)) -> htpy A B f g -> htpy A B g f.
[A, B, f, g, H] inv_htpy A B f g H -->
    x => inv (B x) (f x) (g x) (H x).


(; Definition 9.1.5 III ;)
def concat_htpy : (A : U) -> (B : (El A -> U)) -> (f : El (Pi A B)) ->
   (g : El (Pi A B)) -> (h : El (Pi A B)) -> (H : htpy A B f g) ->
(K : htpy A B g h) -> htpy A B f h.
[A, B, f, g, h, H, K] concat_htpy A B f g h H K -->
    x => concat (B x) (f x) (g x) (h x) (H x) (K x).

(; Definition 9.1.7 I ;)
htpy_left_whisk : (A : U) -> (B : U) -> (C : U) -> (f : MapNondep A B)
   -> (g : MapNondep A B) -> (h : MapNondep B C) ->
    (H : htpy_nondep A B f g) ->
    htpy_nondep A C (comp_fun A B C h f) (comp_fun A B C h g).

(; Definition 9.1.7 II ;)
htpy_right_whisk : (A : U) -> (B : U) -> (C : U) -> (g : MapNondep B C)
    -> (h : MapNondep B C) -> (H : htpy_nondep B C g h) ->
    (f : MapNondep A B) ->
    htpy_nondep A C (comp_fun A B C g f) (comp_fun A B C h f).
```

```
(; Definition 9.2.1 I ;)
def sec : (A : U) -> (B : U) -> (f : (El A -> El B)) -> U.
[A, B, f] sec A B f -->
    Sigma (Pi B (_ => A)) (g => Pi B (x => Id B (f (g x)) x)).

(; Definition 9.2.1 II ;)
def retr : (A : U) -> (B : U) -> (f : (El A -> El B)) -> U.
[A, B, f] retr A B f -->
    Sigma (Pi B (_ => A)) (g => Pi A (x => Id A (g (f x)) x)).

(; Definition 9.2.1 III ;)
def is_equiv : (A : U) -> (B : U) -> (f : (El A -> El B)) -> U.
[A, B, f] is_equiv A B f --> Prod (sec A B f) (retr A B f).

(; Definition 9.3.1 ;)
Eq_Sigma : (A : U) -> (B : (El A -> U)) -> (s : El (Sigma A (B))) ->
    (t : El (Sigma A (B))) -> Type.

(; Definition 9.3.3 ;)
pair_eq_Sigma : (A : U) -> (B : (El A -> U)) -> (x : El A) ->
    (s : El (Sigma A (B))) -> (t : El (Sigma A (B))) ->
    El (Id (Sigma A (B)) s t) -> Eq_Sigma A B s t.


(; Chapter 10 ;)

(; Definition 10.1.1 ;)
def is_contr : (A : U) -> U.
[A] is_contr A --> Sigma A (x => Pi A (c => Id A c x)).

(; Definition 10.2.1 ;)
def ev_pt : (A : U) -> (B : (El A -> U)) -> (a : El A) -> El (Pi A B)
   -> El (B a).
[A, B, a, f] ev_pt A B a f --> f a.

(; Definition 10.3.1 ;)
def fib :
    (A : U) -> (B : U) -> (f : (El A -> El B)) -> (b : El B) -> U.
[A, B, f, b] fib A B f b --> Sigma A (a => Id B (f a) b).

(; Definition 10.3.2 ;)
def Eq_fib : (A : U) -> (B : U) -> (f : (El A -> El B)) -> (y : El B)
   -> El (fib A B f y) -> El (fib A B f y) -> U.

(; Definition 10.3.4 ;)
def is_contr_map : (A : U) -> (B : U) -> (f : El A -> El B) -> U.
[A, B, f] is_contr_map A B f --> Pi B (b => is_contr (fib A B f b)).

(; Definition 10.4.1 ;)
def coherence_is_coherently_invertible : (A : U) -> (B : U) ->
    (f : (El A -> El B)) -> (g : (El B -> El A)) ->
    (G : htpy B (_ => B) (x => f (g x)) (x => x)) ->
    (H : htpy A (_ => A) (x => g (f x)) (x => x)) -> U.

(; Definition 10.4.3 ;)
htpy_nat : (A : U) -> (B : U) -> (f : (El A -> El B)) ->
```

```
    (g : (El A -> El B)) -> (H : htpy A (_ => B) f g) ->
    (x : El A) -> (y : El A) -> (p : El (Id A x y)) ->
    El (Id (Id B (f x) (g y)) (concat B (f x) (g x) (g y) (H x)
    (ap A B g x y p)) (concat B (f x) (f y) (g y) (ap A B f x y p)
    (H y))).

(; Definition 10.4.4 I ;)
def left_unwhisk : (A : U) -> (x : El A) -> (y : El A) -> (z : El A) ->
    (p : El (Id A x y)) -> (q : El (Id A y z)) -> (r : El (Id A y z))
   -> El (Id (Id A y z) (concat A y y z (concat A y x y (inv A x y p)
   p) q) (concat A y y z (concat A y x y (inv A x y p) p) r)) ->
    El (Id (Id A y z) q r).

(; Definition 10.4.4 II;)
def right_unwhisk : (A : U) -> (x : El A) -> (y : El A) -> (z : El A)
   -> (p : El (Id A x y)) -> (q : El (Id A x y)) ->
    (r : El (Id A y z)) -> El (Id (Id A x z) (concat A x y z p r)
    (concat A x y z q r)) -> El (Id (Id A x y) p q).




(; Chapter 11 ;)

(; Definition 11.1.1 ;)
tot : (A : U) -> (x : El A) -> (B : (El A -> U)) -> (C : (El A -> U))
   -> f : (El (Pi A B) -> El (C x)) -> (y : El (Sigma A (B)) ->
    El (Sigma A (C))).

(; Definition 11.1.5 ;)
tot_family : (A : U) -> (B : U) -> (C : (El A -> U)) ->
    (D : (El B -> U)) -> (El (Sigma A (C)) -> El (Sigma B (D))).

(; Definition 11.2.1 ;)
fundemental : (A : U) -> (B : (El A -> U)) -> (a : El A) ->
    (b : El (B a)) -> (x : El A) -> (y : El (B x)) ->
    El (Id (Sigma A B) (pair A B x y) (pair A B a b)).

(; Definition 11.4.1 ;)
def is_emb : (A : U) -> (B : U) -> (f : (El A -> El B)) -> U.
[A, B, f] is_emb A B f -->
    Pi A (x => Pi A (y => is_equiv (Id A x y) (Id B (f x) (f y))
    (ap A B f x y))).

empty : U.
coprod : U -> U -> U.
inl : (A : U) -> (B : U) -> El A -> El (coprod A B).
inr : (A : U) -> (B : U) -> El B -> El (coprod A B).


(; Definition 11.5.2 ;)
def Eq_coprod : (A : U) -> (B : U) -> El (coprod A B) ->
    El (coprod A B) -> U.
(; These are slow. Only uncomment if you are willing to wait a year
[A, B, x, x'] Eq_coprod A B (inl A B x) (inl A B x') --> Id A x x'
[A, B, x, y'] Eq_coprod A B (inl A B x) (inr A B y') --> empty
```

```
[A, B, y, x'] Eq_coprod A B (inr A B y) (inl A B x') --> empty
[A, B, y, y'] Eq_coprod A B (inr A B y) (inr A B y') --> Id B y y'.
;)

(; Definition 11.6.1 ;)
dep_identity_system : (A : U) -> (a : El A) -> (C : (El A -> U)) ->
    (c : El (C a)) -> (B : (El A -> U)) -> (b : El (B a)) ->
    (D : El (Pi A (x => Pi (B x) (_ => C x)))) -> (d : El (C a)) ->
    U.



(; Chapter 12 ;)

(; Definition 12.1.1 ;)
def is_prop : (A : U) -> U.
[A] is_prop A --> Pi A (x => Pi A (y => is_contr (Id A x y))).

(; Definition 12.2.1 I ;)
def is_subtype : (A : U) -> (B : (El A -> U)) -> U.
[A, B] is_subtype A B --> Pi A (x => is_prop (B x)).

(; Definition 12.2.1 II ;)
def is_property : (A : U) -> (B : (El A -> U)) -> U.
[A, B] is_property A B --> is_subtype A B.

(; Definition 12.3.1 ;)
def is_set : (A : U) -> U.
[A] is_set A --> Pi A (x => Pi A (y => is_prop (Id A x y))).


def UU_Set : U.
def UU_Prop : U.
def type_Prop : El UU_Prop -> U.
def type_hom_Prop : El UU_Prop -> El UU_Prop -> U.
[P, Q] type_hom_Prop P Q --> Pi (type_Prop P) (x => type_Prop Q).
T : Type.
suc_T : T -> T.
neg_two : T.



(; Definition 12.4.1 ;)
def is_trunc : (k : T) -> (A : U) -> U.
[A] is_trunc neg_two A --> is_contr A
[A, k] is_trunc (suc_T k) A -->
    Pi A (x => Pi A (y => is_trunc k (Id A x y))).

(; Chapter 13 does not have any definitions ;)



(; Chapter 14 ;)


(; Definition 14.1.1 ;)
```

```
precomp_Prop : (A : U) -> (P : U) -> (hP : El (is_prop P)) ->
    (Q : U) -> (hQ : El (is_prop Q)) -> (El P -> El Q) ->
    (f : (El A -> El P)) -> (El A -> El Q).

(; Required functions ;)
is_prop_all_elements_equal : (A : U) -> (x : El A) -> (y : El A) ->
    (id : El (Id A x y)) -> El (is_prop A).
trunc_Prop : U -> U.
type_trunc_Prop : U -> U.
unit_trunc_Prop : (A : U) -> El A -> El (trunc_Prop A).
all_elements_equal_type_trunc_Prop : (A : U) ->
    (x : El (trunc_Prop A)) -> (y : El (trunc_Prop A)) ->
    El (Id (trunc_Prop A) x y).

(; Definition 14.2.2 ;)
ind_trunc_Prop : (A : U) -> (P : (El (trunc_Prop A) -> U)) ->
    (x : El A) -> El (P (unit_trunc_Prop A x)) ->
    (x : El (trunc_Prop A)) -> (y : El (trunc_Prop A)) ->
    (u : El (P x)) -> (v : El (P y)) ->
    El (Id (P y) (tr (trunc_Prop A) x y (
   all_elements_equal_type_trunc_Prop A x y) P u) v) ->
    (z : El (trunc_Prop A)) -> El (P z).

(; Definition 14.3.1 ;)
def disj_Prop : (P : U) -> (Q : U) -> U.
[P, Q] disj_Prop P Q --> trunc_Prop (coprod P Q).

(; Definition 14.3.3 ;)
def exists_Prop : (A : U) -> (P : (El A -> U)) -> U.
[A, P] exists_Prop A P --> trunc_Prop (Sigma A (x => P x)).

(; Definition 14.4.3 ;)
def is_weakly_constant_map : (A : U) -> (B : U) ->
    (f : (El A -> El B)) -> U.
[A, B, f] is_weakly_constant_map A B f -->
    Pi A (x => Pi A (y => Id B (f x) (f y))).

(; Chapter 15 ;)

(; Definition 15.1.1 ;)
def hom_slice : (X : U) -> (A : U) -> (B : U) ->
    (f : (El A -> El X)) -> (g : (El B -> El X)) -> U.
[X, A, B, f, g] hom_slice X A B f g -->
    Sigma (Pi A (h => B)) (h => Pi A (x => Id X (f x) (g (h x)))).

(; Definition 15.1.2 ;)
def comp_hom_slice : (X : U) -> (A : U) -> (B : U) -> (C : U) ->
    (f : (El A -> El X)) -> (g : (El B -> El X)) ->
    (h : (El C -> El X)) -> El (hom_slice X B C g h) ->
    El (hom_slice X A B f g) -> El (hom_slice X A C f h).

def emb : U -> U -> U.
def map_emb : (A : U) -> (B : U) -> El (emb A B) -> (El A -> El B).

(; Definition 15.1.2 ;)
precomp_emb : (X : U) -> (A : U) -> (f : (El A -> El X)) ->
```

```
    (B : U) -> (i : (El (emb B X))) ->
    El (hom_slice X A B f (map_emb B X i)) -> (C : U) ->
    (j : (El (emb C X))) -> El (hom_slice X B C (map_emb B X i)
    (map_emb C X j)) -> El (hom_slice X A C f (map_emb C X j)).

(; Definition 15.1.5 ;)
def im : (A : U) -> (X : U) -> (f : (El A -> El X)) -> U.
[A, X, f] im A X f --> Sigma X (x => trunc_Prop (fib A X f x)).

(; Definition 15.2.1 ;)
def is_surjective : (A : U) -> (B : U) -> (f : El A -> El B) -> U.
[A, B, f] is_surjective A B f --> Pi B (y => trunc_Prop (fib A B f y)).

(; Definition 15.3.1 ;)
def powerset : (X : U) -> U.
```

A.2 .dkgf content

The following is the content of the **.dkgf** file used to link the formalized Dedukti code presented in Appendix A.1.

```
#MACRO \newcommand{\universe}[0]{U}

U : "the universe"
#MACRO \newcommand{\element}[1]{#1}
El  : "element of X"

#MACRO \newcommand{\Pitype}[2]{\Pi_{x:#1}#2(x)}
#DROP Pi 0
Pi : "the dependent pair of X and Y" | \Pitype


#DROP Sigma 0
#MACRO \newcommand{\sigmatype}[2]{\Sigma_{#1} #2}
Sigma : "the sigma type of X and Y" | \sigmatype

#DROP pr1 2
pr1 : "the first element of X"

#DROP pr2 2
pr2 : "the second element of X"

Prod : "product of X and Y" | \product

#DROP pair_prod 2
pair_prod : "the pair of X and Y"

#MACRO \newcommand{\product}[2]{#1 \times #2}
prod : "the product of X and Y" | \product

#MACRO \newcommand{\eq}[2]{#1 = #2}
#DROP Id 1
Id : "the equality type between X and Y" | \eq
```

```
#DROP tr 4
#MACRO \newcommand{\transport}[2]{\text{tr}_{#1}(\alpha, #2)}
tr : "transport X on Y" | \transport

#DROP concat 4
#MACRO \newcommand{\concat}[2]{#1 \cdot #2}
concat : "the concatenation of X and Y" | \concat

#DROP inv 3
#MACRO \newcommand{\inv}[1]{\text{inv}(#1)}
inv : "the inverse of X" | \inv

#DROP comp_fun 3
#MACRO \newcommand{\compfun}[2]{#1 \circ #2}
comp_fun : "composition." | "the composition of X and Y" | \compfun

#DROP 1 SigmaMap
#MACRO \newcommand{\sigmamap}[2]{\Sigma_{#2} #1}
SigmaMap : "the sigma type of X and Y" | \sigmamap

#DROP ap 5
#MACRO \newcommand{\ap}[1]{\text{ap}_{#1}}
ap : "path action." | "the action on X" | \ap

#DROP pair 2
#MACRO \newcommand{\pair}[2]{P (#1,#2)}
pair : "the pair of X and Y" | \pair

#MACRO \newcommand{\copro}[2]{#1 + #2}
coprod : "the coproduct of X and Y" | \copro


#DROP id_fun 1
#MACRO \newcommand{\idfun}[1]{\text{id}_{#1}}
id_fun : "the identity function on X" | \idfun

#DROP htpy 2
#MACRO \newcommand{\homotopy}[2]{#1 \sim #2}
htpy        : "X is homotopic to Y" | "the homotopy between X and Y" |
   \homotopy

#DROP htpy_nondep 2
htpy_nondep : "X is homotopic to Y" | "X and Y are homotopic" | "
   homotopy between X and Y" | \homotopy

#MACRO \newcommand{\map}[2]{#1 \rightarrow #2}
Map         : "map from X to Y"
MapNondep   : "map from X to Y"

#DROP refl_htpy 2
#MACRO \newcommand{\reflhtpy}[1]{\text{refl-htpy}_{#1}}
refl_htpy   : "reflexivity." | "the reflexive homotopy of X" | \
   reflhtpy

#DROP inv_htpy 4
#MACRO \newcommand{\invhtpy}[1]{\text{inv-htpy}_{#1}}
```

```
inv_htpy    : "inverse." | "the inverse homotopy of X" | \invhtpy

#DROP concat_htpy 5
#MACRO \newcommand{\conhtpy}[2]{\text{concat-htpy}(#1,#2)}
concat_htpy : "concatenation." |  "the concatenation of X and Y" | \
   conhtpy

#MACRO \newcommand{\leftwhisk}[2]{#1 \cdot #2}
#DROP htpy_left_whisk 5
htpy_left_whisk  : "the left whiskering of X by Y" | \leftwhisk
#MACRO \newcommand{\rightwhisk}[2]{#1 \cdot #2}
#DROP htpy_right_whisk 5
htpy_right_whisk : "the right whiskering of X by Y" | \rightwhisk

#DROP sec 2
#MACRO \newcommand{\sect}[1]{\text{sec} (#1)}
sec  : "section of X" | \sect
#DROP retr 2
#MACRO \newcommand{\retr}[1]{\text{retr} (#1)}
retr : "retraction of X" | \retr

#MACRO \newcommand{\isequiv}[1]{\text{is-equiv(#1)}}
#DROP is_equiv 2
is_equiv : "equivalence." | "X is an equivalence" | \isequiv

#MACRO \newcommand{\equivalance}[2]{#1 \simeq #2}
equiv    : "equivalence from X to Y" | \equivalance

#DROP map_equiv 2
map_equiv            : "the underlying map of X"

#DROP is_equiv_map_equiv 2
is_equiv_map_equiv   : "the proof that the map of X is an equivalence"

#DROP is_equiv_has_inverse 2
is_equiv_has_inverse : "the proof that X is an equivalence from an
   inverse"

#DROP Eq_Sigma 2
#MACRO \newcommand{\sigmaeq}[2]{\text{eq}_{\Sigma} (#1, #2)}
Eq_Sigma           : "the component equality between X and Y" | \
   sigmaeq


#DROP pair_eq_Sigma 5
pair_eq_Sigma      : "dependent pair equivalence."

#MACRO \newcommand{\iscontr}[1]{\text{is-contr}(#1)}
#DROP is_contr 0
is_contr : "contractible." | "X is contractible" | \iscontr
#DROP center 1
center : "center." | "X has a center"

#DROP ev_pt 2
#MACRO \newcommand{\ev}[2]{\text{ev-pt}}
ev_pt : "the evaluation of X at Y"
```

```
#DROP fib 2
#MACRO \newcommand{\fibration}[2]{\text{fib}_{#1}(#2)}
fib : "the fiber of X over Y" | \fibration


#DROP Eq_fib 4
Eq_fib : "the fiber equality of X and Y"

#DROP is_contr_map 2
is_contr_map : "X is a contractible map"

#DROP coherence_is_coherently_invertible 4
coherence_is_coherently_invertible : "coherence invertible."

htpy_nat : "natural homotopy."

#DROP left_unwhisk 4
left_unwhisk : "left unwhiskering."

#DROP right_unwhisk 4
right_unwhisk : "right unwhiskering."

#DROP tot 6
tot : "Total-space." | "the total space"

#DROP tot_family 6
tot_family : "Total-family."

#DROP fundemental 6
fundemental : "identity system."

#DROP is_emb 5
#MACRO \newcommand{\emb}[1]{\text{is-emb}(#1)}
is_emb : "embedding." | "the embedding of X" | \emb

#DROP Eq_coprod 2
#MACRO \newcommand{\eqcoprod}[2]{\text{Eq-copr}_{#1, #2}}
Eq_coprod : "observational equality between X and Y" | \eqcoprod

#DROP is_prop 0
#MACRO \newcommand{\isprop}[1]{\text{is-prop}(#1)}
is_prop : "proposition." | "X is a proposition" | \isprop

#MACRO \newcommand{\issubtype}[2]{\text{is-subtype}_{#2}(#1)}
is_subtype : "the subtype of Y over X"


#MACRO \newcommand{\isproperty}[2]{\text{is-property}(#1)}
is_property : "Y is a property of X" | \isproperty


#MACRO \newcommand{\isset}[1]{\text{is-set}(#1)}
is_set : "X is a set" | \isset
```

```
UU_Set : "the set universe"

UU_Prop : "the property universe"

#MACRO \newcommand{\typeProp}[1]{\text{type-prop}(#1)}
type_Prop : "type property of X" | \typeProp

#MACRO \newcommand{\typeHomProp}[2]{\text{type-hom-prop}(#1, #2)}
type_hom_Prop : "type property of X and Y" | \typeHomProp


#MACRO \newcommand{\trunc}[0]{\mathbb{T}}
T : "truncation level" | \trunc
#MACRO \newcommand{\truncsucc}[0]{\text{succ}_{\mathbb{T}}}
suc_T : "successor of X" | \truncsucc

#MACRO \newcommand{\truncbase}[0]{\text{-2}_{\mathbb{T}}}
neg_two : "base level" | \truncbase


#MACRO \newcommand{\istrunc}[2]{\text{is-trunc}_{#1}(#2)}
is_trunc : "X is a truncation of Y" | \istrunc

#DROP hom_slice 3
#MACRO \newcommand{\homslice}[2]{\text{hom}_X(#1, #2)}
hom_slice : "the slice of X and Y" | \homslice

#DROP precomp_Prop 6
precomp_Prop : "X is a proposition truncation"

#DROP is_prop_all_elements_equal 3
is_prop_all_elements_equal : "X holds for propositions"

#MACRO \newcommand{\truncprop}[1]{||#1||}
trunc_Prop : "the truncation proposition of X" | \truncprop

#MACRO \newcommand{\trunctypeprop}[1]{\text{type-trunc-prop}(#1)}
type_trunt_Prop : "type truncation of proposition X" | \trunctypeprop

#MACRO \newcommand{\truncpropunit}[2]{||#1||_{0}}
unit_trunc_Prop : "unit truncation proposition of X and Y" | \
   truncpropunit
all_elements_equal_type_trunc_Prop : "X and Y are equal"

ind_trunc_Prop : "induction principle truncation."

#MACRO \newcommand{\disjtrunc}[2]{#1 \lor #2}
disj_Prop : "disjunction of X and Y" | \disjtrunc

#MACRO \newcommand{\existtrunc}[2]{\exists_{x : #1} #2(x)}
exists_Prop : "existential quantification of X and Y" | \existtrunc

#DROP is_weakly_constant_map 2
#MACRO \newcommand{\weakly}[1]{\text{is-weakly-constant} (#1)}
is_weakly_constant_map : "weakly-constant." | "constant map X"
```

```
#DROP comp_hom_slice 7
comp_hom_slice : "morphism."

#MACRO \newcommand{\emb}[2]{#1 \hookrightarrow #2}
emb : "embedding of X and Y" | \emb

#DROP map_emb 2
#MACRO \newcommand{\mapemb}[1]{\text{map-emb}(#1)}
map_emb : "map embedding." | "map embedding of X" | \mapemb

precomp_emb : "universal property."


#MACRO \newcommand{\image}[1]{\text{im}(#1)}
#DROP im 2
im : "image of X" | \image

#DROP is_surjective 2
#MACRO \newcommand{\surjective}[1]{\text{is-surj}(#1)}
is_surjective : "X is surjective" | \surjective

#MACRO \newcommand{\powerset}[1]{\text{P}_{U}(#1)}
powerset : "power set of X" | \powerset

empty : "empty type" | \emptyset

#MACRO \newcommand{\inl}[1]{\text{inl}(#1)}
#DROP inl 2
inl : "left X" | \inr

#MACRO \newcommand{\inr}[1]{\text{inr}(#1)}
#DROP inr 2
inr : "right X" | \inr

dep_indentity_system : "dependent identity."
type_trunc_Prop : "typetruncprop."
```