Delft University of Technology

Programming Languages Group

Termination Checking in Agda-Core

Master's Thesis

submitted in partial fulfillment of the requirements
for the degree of

Master of Science

Author:
Arthur Jacques
Student ID: 5446988

Supervisor:
Prof. Jesper Cockx

Daily Co-supervisor:
Nathaniel Burke

June 23, 2026

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Jesper Cockx, for his guidance throughout this project. My meetings with him were always insightful, and many of the ideas presented in this work originated from our discussions. His expertise in the field, thoughtful feedback, and ability to identify promising directions were invaluable to the success of this thesis.

I would also like to thank my daily co-supervisor, Nathaniel Burke, for his constant support throughout the project. He was always willing to help with the many challenges I encountered while working in Agda, and his deep technical knowledge often enabled us to quickly identify and resolve difficult problems. His availability, patience, and guidance were instrumental in helping me make steady progress.

AI Usage Statement

During the preparation of this thesis, I made limited use of generative artificial intelligence tools as research and writing aids.

For the implementation work, I primarily used Claude Sonnet 4.6 to help understand concepts related to Agda and dependent type theory, and occasionally to assist in identifying potential issues in proofs and implementations. Suggestions produced by the system were always reviewed, evaluated, and adapted before being incorporated into the work.

For the writing process, I primarily used GPT-5.5 to obtain feedback on writing style, clarity, and overall structure. The system was also used to provide comments on whether the flow and organization of sections were coherent and, in some cases, to assist with the rewriting of paragraphs. All final wording, technical content, interpretations, and conclusions remain my own responsibility.

The AI tools were not used to generate original research contributions, proofs, formalizations, or results presented in this thesis. Their role was limited to explanation, feedback, editing assistance, and debugging support.

Abstract

Dependently typed languages and proof assistants such as Agda and Rocq improve the reliability of mathematical proofs and programs by representing logical propositions as types and proofs as programs inhabiting those types. Through the Curry–Howard correspondence, typechecking becomes proof checking, allowing correctness guarantees to be established by construction.

However, these guarantees ultimately rely on the correctness of the language implementation itself. Components such as typecheckers, conversion checkers, and termination checkers are typically implemented as algorithms whose correctness must be trusted independently from the theory they are intended to enforce.

Agda Core is a core language for Agda implemented in Agda itself, where language judgements are represented directly as dependent types and checking procedures become programs constructing evidence of those judgements. This thesis investigates how the same methodology can be applied to termination checking.

Termination checking is a fundamental component of dependently typed languages, since unrestricted recursion may compromise normalization and logical consistency. We explore how termination criteria can be represented as formal specifications together with executable proof-search procedures producing explicit certificates that the criteria hold.

We first study the guard condition, a structural recursion criterion based on descending recursive arguments, and implement a certified checker producing explicit evidence that the criterion holds. We then investigate the size change principle, a stronger termination criterion capable of handling a wider class of recursive and mutually recursive definitions. We formalize the corresponding rules in the same framework and discuss the challenges involved in constructing a fully executable checker for them.

More generally, this work explores the separation between declarative specifications of termination criteria and the algorithms used to search for certificates satisfying them. By expressing termination arguments directly in the type theory of the implementation language, the resulting infrastructure becomes simultaneously executable, inspectable, and partially verified.

Chapter 1

Introduction

Languages such as Agda and Rocq have been developed with the idea of enabling their users to automatically check proofs of propositions, thereby improving the process of proving theorems for mathematicians and scientists in general through increased reliability. This automatic proof checking is enabled by the Curry-Howard correspondence, which establishes a link between logic and computation. It allows one to represent propositions as types and programs of that type as proofs of the corresponding proposition. One can then represent complex proposition such as universal (for all) and existential (there exists) quantification over values using Π and Σ types respectively. Typecheckers effectively become proof checkers, where checking that a program has a given type corresponds to verifying a proof of a given proposition.

However, in languages that support recursion, well-typedness is not the only condition for a term to be well-formed; the term must also be terminating. Under the Curry–Howard correspondence, recursive programs correspond to proofs by induction, and unrestricted recursion can break logical consistency by allowing the construction of non-normalizing proof terms. In particular, recursive definitions can make it possible to derive an inhabitant of the empty type. Because of this, proof assistants and dependently typed languages must check that recursive definitions terminate.

Unfortunately, as demonstrated by Turing, determining whether an arbitrary program terminates is undecidable [23]. As a result, dependently typed languages must rely on conservative termination criteria that reject some terminating programs while never accepting programs that violate the criterion. Different such criteria have been developed, such as the guard condition [12] used in Rocq and the size change principle [13] used in Agda. A more recent approach integrates termination checking directly into the type system through sized types [1], although known issues remain in current implementations [3]. The program responsible for deciding whether a definition satisfies a chosen criterion is called the termination checker.

The practical purpose of dependently-typed languages such as Agda and Rocq is twofold. In addition to their use for writing proofs, they allow programmers to give refined types to programs, and hence give those programs correctness guarantees stronger than in languages with simpler type systems, once they pass the typechecker (and termination checker). In both cases, one has to trust the implementation of the language. However, those implementations are just software, and hence prone to bugs. The applications of these systems are precisely those where high trustworthiness is very important, as bugs could allow for the validation of erroneous proofs. Even if the scenarios in which such a bug could play out is niche, it could be exploited by a malicious user to give a bugged proof to an erroneous statement or type. Hence verifying the correctness of such languages is a primary concern.

1.1 Language Verification and Agda-Core

A solution to the problem of untrustworthiness in language implementation is the adoption of a core language. A core language is a subset of the constructs of a target language, to which all features of the target languages can be elaborated to. This approach has been adopted notably in Rocq and Lean [10]. The main goal of this is to reduce the code one has to trust to only the core language, and the elaboration of the other features into the core ones.

Another approach is to formally specify the rules of the language inside a proof assistant itself, and derive executable infrastructure from those specifications. Rather than implementing typechecking or other components as ordinary algorithms returning booleans, one encodes the rules of the system as dependent types. A checker then becomes a program attempting to construct evidence that the corresponding judgement holds.

MetaRocq [20] follows this methodology for Rocq's core language. It formalizes substantial parts of Rocq's metatheory and studies properties such as subject reduction and confluence. Agda Core [14] is a related project for Agda, but with a different scope and philosophy. Rather than proving metatheoretical properties of the language, Agda Core focuses on expressing the rules of the system directly as dependent types and deriving infrastructure that respects those rules.

Historically, Agda was not designed around a small explicit core language. Agda Core therefore defines and implements such a core language from scratch, together with certified typechecking and conversion checking infrastructure written in Agda itself.

1.2 Termination Checking as Proof-Relevant Specification

This thesis investigates how termination checking can be integrated into the methodology used by Agda Core.

Traditional termination checkers are implemented as algorithms which traverse programs and either accept or reject them according to some criterion. In Agda Core, termination criteria are instead represented directly as dependent types describing valid termination arguments, while checkers become programs attempting to construct evidence inhabiting those types.

This raises an important question: how naturally do existing termination criteria integrate into such a framework?

Some criteria appear to admit direct structural representations as inductive specifications, while others rely on more global operational reasoning that becomes substantially more difficult to express. This thesis investigates that distinction through two case studies.

We first study a simplified version of the guard condition and derive executable proof-producing checkers for it. We then investigate the size change principle, a stronger criterion capable of handling a wider class of recursive definitions, and analyse the additional difficulties involved in deriving certified executable infrastructure for it.

1.3 Contributions

The contributions of this thesis are the following:

- We investigate how termination criteria can be represented as proof-relevant specifications in Agda Core using dependent types.
- We first formalize a simplified guard condition and derive executable checkers constructing explicit evidence that functions satisfy the criterion. We show that the criterion can be represented relatively naturally as a proof-relevant specification using dependent types.
- We formalize a specification of the size change principle in the same framework. While its declarative specification can also be expressed inside Agda Core, deriving an executable certified

checker exposes additional difficulties related to graph exploration and reasoning about infinitely many call cycles.

In chapter 2, We cover some background concerning dependent types, the guard condition, size change termination and Agda Core, necessary for understanding the contributions. In chapter 3, we cover the implementation of the contributions. In chapter 4, we review some related work. We conclude in chapter 5.

Chapter 2

Background

2.1 Dependent types

Let us first see how dependent types work more in detail. In a simply typed language, the type of a term is independent of values. A function `reverse : List A → List A` tells us the output is a list, but nothing about its length. With dependent types, types can *depend on values*, which allows more precise specifications. For instance, one can define `Vec A n`, a list of exactly `n` elements of type `A`, where `n` is an ordinary natural number value:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Here `Vec A` is not a type but a *family* of types, indexed by a natural number. `Vec A zero` and `Vec A (suc zero)` are distinct types. This lets us give `reverse` the more informative type `Vec A n → Vec A n`, ruling out at the type level any implementation that changes the length. Similarly, a safe head function can have type `Vec A (suc n) → A`, making it impossible to call on an empty vector — no runtime check needed.

The key type former enabling this is the Π *type* $\Pi(x : A).\ B(x)$, which generalises the function type $A \to B$ by allowing the return type to mention the argument. A term of type $\Pi(x : A).\ B(x)$ is a function that, given $a : A$, returns a result of type $B(a)$. The ordinary arrow type is the special case where B does not mention x. In Agda, dependent function types are written using the notation `(x : A) → B x`. So `Vec A n → Vec A n` is shorthand for $\Pi(n : \mathbb{N}).$ `Vec A n` $\to$ `Vec A n`, and the return type genuinely depends on the value of `n` passed in.

Types as Propositions

The expressive power of Π types goes beyond precise data structure specifications. One can also form types that directly express logical statements. Consider the *propositional equality* type, written $a \equiv b$ for $a\ b : A$:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a

infix 4 _≡_

zeroeqzero : zero ≡ zero
zeroeqzero = refl {ℕ} {zero}
```

The type `a ≡ b` is a proposition: it is inhabited (has a term) exactly when `a` and `b` are equal. Its only constructor, `refl`, produces a proof of `a ≡ a`. A term of type `zero + n ≡ n` is therefore not a computation that *returns* true or false; it is a *proof* that those two expressions denote the same value.

This is the **Curry-Howard correspondence**: types correspond to propositions, and terms correspond to proofs. Function types $A \to B$ correspond to implication; Π types $\Pi(x : A).\ B(x)$ correspond to universal quantification; the Sigma type $\Sigma(x : A).\ B(x)$ (dependent pairs) corresponds to existential quantification. The empty type $\bot$ — with no constructors and hence no inhabitants — corresponds to falsity, and $\neg A$ is defined as $A \to \bot$.

As a concrete example, we can prove that minimum of a natural number and itself is itself by constructing a term of type $\Pi(n : \mathbb{N}).\ min\ n\ n \equiv n$:

```
min : ℕ → ℕ → ℕ
min zero _    = zero
min _    zero = zero
min (suc m) (suc n) = suc (min m n)

min-id : (x : ℕ) → min x x ≡ x
min-id zero = refl
min-id (suc x) = cong suc (min-id x)
```

The typechecker verifies that this term is well-typed, and in doing so, it verifies the proof. Writing a program and writing a proof become the same activity.

However, as mentioned in the introduction, well-typedness is not the only condition to the well-formedness of a term. Terms that use recursion to do induction, such as the one above, could be ill-formed. An example of a term that would typecheck but be problematic is the following:

```
wrong : zero ≡ suc zero
wrong = wrong
```

This definition is well-typed, but it is not well-formed, as its execution will infinitely recurse. This is why a termination checker is needed.

2.2 The guard condition

The first termination condition (the criterion used by the termination checker) we will review is the guard condition. It can be summarized as follows: at least one argument must be strictly descending (structurally smaller) across all recursive calls in the function's body. A value is structurally smaller if it is a subterm obtained by pattern matching — after a match on `suc n`, n is smaller than `suc n`.

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
(suc n) + m = suc (n + m)
```

In this example, the only recursive call is done on a smaller first argument: n is smaller than suc n.

```
fib : ℕ → ℕ
fib zero        = zero
fib (suc zero) = suc zero
fib (suc (suc n)) = fib n + fib (suc n)
```

Here there are two recursive calls (both on the last line), and both are descending in the same argument, namely the only argument of the function.

```
bad : ℕ → ℕ
bad zero = zero
bad (suc n) = bad n + bad (suc (suc n))
```

This is a negative example: the second recursive call on the last line is done on an argument which is not structurally smaller, making the function non-terminating.

```
bad2 : ℕ → ℕ → ℕ
bad2 zero zero = zero
bad2 m    (suc n) = bad2 m n
bad2 (suc m) zero = bad2 m zero
```

This is an example of a program that is terminating but not accepted by the guard condition. There are two recursive calls, one descending in the first argument and the other in the second. Accepting this would require a stronger criterion, such as one based on lexicographic order or the size change principle, as we will see later.

2.3 The Size Change Principle

The guard condition presented previously is a simple and effective termination criterion, but it remains limited in expressiveness. In particular, it analyses recursive calls individually and therefore rejects many programs whose termination only becomes apparent when several calls are considered together.

A more powerful criterion is the *size change principle* (SCP), introduced by Lee, Jones and Ben-Amram [13]. SCP naturally supports mutually recursive definitions and is the criterion employed by Agda itself. Rather than inspecting recursive calls in isolation, it reasons globally about how arguments evolve across entire chains of function calls.

2.3.1 Infinite Descent

The original formulation of SCP is stated in terms of infinite executions.

Consider an infinite sequence of function calls

$$f_0 \to f_1 \to f_2 \to \cdots$$

representing a hypothetical non-terminating execution of a program. SCP tracks how the arguments of these functions evolve throughout such a sequence.

For every call, one records whether an argument becomes strictly smaller, remains unchanged, or whether no relationship can be established. The crucial observation is that an infinite execution can only exist if it is possible to follow arguments indefinitely through the call sequence.

SCP therefore requires that every infinite execution contain an *infinitely descending thread*

$$x_0 > x_1 > x_2 > \cdots$$

where each x_i is an argument value occurring during the execution.

Since the decrease relation is based on the subterm ordering, which is well-founded, such an infinitely descending sequence cannot exist. Consequently, any infinite call sequence satisfying this property is impossible.

The size change principle can therefore be stated as follows.

> A program is terminating if every infinite call sequence contains an infinitely descending thread.

While this formulation captures the intuition behind SCP, it is not directly suitable for implementation. An implementation cannot explicitly reason about all possible infinite executions. Instead, it must derive finite information from the program.

2.3.2 Call Graphs and Size-Change Matrices

To obtain a finite representation, SCP analyses the call graph of a program.

For each call site, one records how the arguments of the callee relate to the parameters of the caller. These relationships are represented by a *size-change matrix*, whose entries are one of three values:

- $<$: the argument of the callee is a strict subterm of the corresponding parameter of the caller;
- $=$: the argument of the callee is identical to the corresponding parameter of the caller;
- ∞ : no size relationship is known.

These matrices are then assembled into a call graph. Each node represents a function and each edge represents a function call labelled by its corresponding size-change matrix.

The matrix associated with a path of calls is obtained by composing the matrices associated with each individual edge. Composition is defined by

$\times$	$<$	$=$	∞
$<$	$<$	$<$	∞
$=$	$<$	$=$	∞
∞	∞	∞	∞

and extended pointwise to matrices.

Intuitively, composition accumulates information across multiple calls. This is precisely what allows SCP to detect decreases that are invisible when examining individual recursive calls in isolation.

2.3.3 Cycles and Idempotent Cycles

The semantic statement of SCP concerns infinite call sequences. However, any infinite execution must repeatedly revisit the same functions.

Consequently, implementations study cycles in the call graph. A cycle represents behaviour that may be repeated arbitrarily many times during an execution. By composing the matrices associated with the calls in a cycle, one obtains the cumulative effect of traversing the cycle once.

A cycle is considered terminating if its composed matrix contains a strict decrease on the diagonal. Such an entry indicates that some parameter becomes strictly smaller than itself after following the cycle.

At first sight, one might expect that every cycle in the transitive closure of the call graph must be checked. The SCP paper shows that this is unnecessary.

Instead, it is sufficient to consider *idempotent cycles*. A cycle is idempotent if its associated matrix M satisfies

$$M \times M = M.$$

Intuitively, an idempotent cycle represents a stable behaviour of the call graph. Once such a cycle has been reached, traversing it again cannot reveal any new size-change information.

The reason this suffices is that any infinite execution must eventually exhibit stable behaviour. Using a Ramsey-theoretic argument, the SCP paper shows that from any infinite call sequence one can extract an idempotent cycle occurring infinitely often. Therefore, if every idempotent cycle contains a strict decrease on its diagonal, every infinite execution must contain an infinitely descending thread.

This yields the operational formulation of SCP:

> A program is terminating if every idempotent cycle in the transitive closure of its call graph contains a strict decrease on its diagonal.

2.3.4 Examples

First, consider the following mutually recursive program.

```
mutual
  isEven : ℕ → Bool
  isEven zero = true
  isEven (suc n) = isOdd n

  isOdd : ℕ → Bool
  isOdd zero = false
  isOdd (suc n) = isEven n
```

The call graph contains two nodes and two edges:

$$\texttt{isEven} \xrightarrow{<} \texttt{isOdd} \xrightarrow{<} \texttt{isEven}.$$

Composing the two matrices yields

$$(<).$$

The resulting cycle contains a strict decrease on the diagonal and is therefore terminating. Since repeated compositions preserve this decrease, all idempotent cycles are terminating and SCP accepts the program.

Now consider the non-terminating function

```
bad : ℕ → ℕ
bad zero = zero
bad (suc n) = bad n + bad (suc (suc n))
```

The first recursive call yields

$$M_1 = (<) .$$

while the second yields

$$M_2 = (\infty) .$$

The call graph therefore contains two self-loops.

Repeatedly following the second loop yields the idempotent matrix

$$(\infty) .$$

Since this matrix contains no strict decrease on its diagonal, the corresponding idempotent cycle is non-terminating. SCP therefore rejects the program.

Finally, consider

```
bad2 : ℕ → ℕ → ℕ
bad2 zero zero = zero
bad2 m    (suc n) = bad2 m n
bad2 (suc m) zero = bad2 m zero
```

This function is terminating but rejected by the simple guard condition because different recursive calls decrease different arguments.

The corresponding matrices are

$$A = \begin{pmatrix} = & \infty \\ \infty & < \end{pmatrix} .$$

and

$$B = \begin{pmatrix} < & \infty \\ \infty & = \end{pmatrix} .$$

Neither matrix decreases both arguments simultaneously. Consequently, an analysis based solely on individual recursive calls cannot establish termination.

The SCP analysis instead considers the transitive closure of the call graph. In this example, the matrices

$$A \times A = A$$

and

$$B \times B = B$$

show that both A and B are already idempotent. Furthermore,

$$A \times B = \begin{pmatrix} < & \infty \\ \infty & < \end{pmatrix} ,$$

and

$$B \times A = \begin{pmatrix} < & \infty \\ \infty & < \end{pmatrix},$$

which is itself idempotent. Thus the transitive closure contains four relevant idempotent cycles: A, B, AB, and BA.

The SCP criterion requires that each of these idempotent cycles contain a strict decrease on its diagonal. Matrix A decreases the second argument, matrix B decreases the first argument, and both AB and BA decrease both arguments. Consequently, all idempotent cycles are terminating, and SCP accepts the program.

Intuitively, repeated applications of the first clause eventually reduce the second argument to `zero`; once this happens, the second clause reduces the first argument. Since both arguments are natural numbers, this process must terminate.

This example illustrates the additional power of SCP. While the guard condition inspects recursive calls individually, SCP reasons about the cumulative effect of chains of calls and therefore accepts a substantially larger class of terminating programs.

2.4 Agda Core

2.4.1 Syntax

Agda Core is meant to be a core language for Agda, developed in Agda itself for the sake of using Agda's proof checking capabilities. The syntax looks at follows:

```
data Term a where
  TVar  : (x : NameIn a) → Term a
  TDef  : (d : NameIn defScope) → Term a
  TData : (d : NameData)
        → (TermS a (dataParScope d))
        → (TermS a (dataIxScope d))
        → Term a
  TCon  : {d : NameData} (c : NameCon d)
        → (TermS a (fieldScope c)) → Term a
  TLam  : (@0 x : Name) (v : Term (a ▶ x)) → Term a
  TApp  : (u : Term a) (v : Term a) → Term a
  TProj : (u : Term a) (x : NameIn defScope) → Term a
  TCase : {@0 x : Name}
        → (d : NameData)
        → Singleton (dataIxScope d)
        → (u : Term a)
        → (bs : Branches a d (AllNameCon d))
        → (m : Type (a ◀▶ dataIxScope d ▶ x))
        → Term a
  TPi   : (@0 x : Name) (u : Type a) (v : Type (a ▶ x)) → Term a
  TSort : Sort a → Term a
  TLet  : (@0 x : Name) (u : Term a) (v : Term (a ▶ x)) → Term a
  TAnn  : (u : Term a) (t : Type a) → Term a
```

Core language: Simpler features

Agda Core is a core language for Agda, which means it has a more minimal set of features than Agda has, and other features are meant to be elaborated to the ones in Agda Core. An example of a complex feature which is present in Agda but not in Agda Core is left-hand side pattern matching. This is not present in Agda Core, and instead must be elaborated to a TCase expression, which has a branch for all constructors of the datatype of the term being pattern matched on. This more closely matches operational semantics, making it more suited for a core language (in Agda, the left-hand side pattern matching is transformed into a case tree as intermediate syntax [7]).

Variable Representation: Scopes as the Best of Both Worlds

Each term is indexed over a scope, which is a list of Name (which is an alias for String). This is the result of a deliberate design choice to work with well-scoped terms throughout the implementation. When implementing a language with binders, there are several common approaches for representing variables. The most common is de Bruijn indices[4], where each variable is represented as a natural number denoting how many binders must be crossed to reach its binding site. While this approach makes α-equivalence trivial (two terms are α-equivalent if and only if they are structurally identical), it comes at the cost of being less readable. At the other extreme, one can use plain string names for variables, which are much easier to read, but offer no guarantees against variable capture or name clashes. Scopes offer a middle ground that captures the strengths of both approaches. Variable names remain strings for readability while each term is statically indexed over the set of names that are in scope at that point in the program. This means that well-scopedness is enforced by the type system itself: it is impossible to construct a term that refers to a name not present in its scope index, and no runtime checks are needed. The result is syntax that is both legible and correct by construction. As an example of how this shows up in the code, We can look at the TLam constructor, which expects a Name (string) x and a Term indexed over the scope extended with x, which implies that the body of the lambda could have x appear as a named variable. Then for TVar, NameIn a consists of a Name and a proof that that name is in a

Erasure notation

In TLam, TLet and TPi, we can see that the name of the variable name is preceded by @0. This notation means that the annotated parameter can only be used on types, and cannot be used computationally [2]. This is important because Agda Core relies on **Agda2hs**[8, 14] to compile the Agda code to haskell and thus have a runnable typechecker, and annotated terms will not be added to the compiled code. In this case, the names are not important because the compiled code makes use of de Bruijn indices [4]. NameIn have their associated name erased as well, and the proof of presence in the scope gets turned into its corresponding de Bruijn index. These erasure notations are used extensively in the rules (for typing, conversion and termination) of Agda Core.

2.4.2 Typing rules and typechecker

We can now see how Agda Core defines typing rules as indexed datatypes. But first we need to be able to represent types and context:

```
record Type a where
  inductive; no-eta-equality; pattern
  constructor El
  field
    typeSort : Sort a
    unType : Term a
```

Type is just a wrapper around its field unType, which is a term representing the type, as well as the sort of that type. In Agda, every type lives at a particular level in an infinite hierarchy of universes, where each universe is itself a type in the next one up. Tracking the sort is necessary to know which level of this hierarchy a given type inhabits [17].

```
data Context : @0 Scope Name → Set where
  CtxEmpty : Context mempty
  CtxExtend : Context a → (@0 x : Name) → Type a → Context (a ▶ x)
```

A Context is a type indexed by a scope, which associates each name bound in the scope with its respective type.

```
data TyTerm (@0 Γ : Context a) :
  @0 Term a → @0 Type a → Set
```

```
syntax TyTerm Γ u t = Γ ⊢ u : t
```

TyTerm is stating that under context Γ, the term u has type t. Then each typing rule can be represented as a constructor to the type TyTerm. For example the rule for lambda can be written as follows:

```
TyLam :
  Γ , x : a ⊢ u : b
  --------------------------------
  → Γ ⊢ TLam x u : El k (TPi x a b)
```

This can be explained as: To inductively build a proof that the context Γ recognizes a TLam constructor with named variable x and body u to have as type a Π type that binds x of type a, with b as the type of its body (possibly depending on x), we need to provide a proof that the context Γ extended with x typed a recognizes that u has type b.

Then the typechecker is built as a monadic program trying to derive a proof for the well-typedness of a term. The signature would be:

```
checkType : ∀ (Γ : Context a) (u : Term a)
            (ty : Type a) → TCM (Γ ⊢ u : ty)
```

This illustrates the methodology followed throughout Agda Core. Rather than implementing typing as a boolean-valued algorithm, typing derivations themselves are represented as inductive structures, and the typechecker becomes a program attempting to construct inhabitants of those structures. A successful run of the checker therefore not only accepts a term, but also produces explicit evidence that the typing judgement holds.

This separates the specification of the typing rules from the algorithm used to derive them. The rules themselves are represented declaratively as dependent types, while the checker merely attempts to construct objects inhabiting those types. Correctness arguments can therefore be stated primarily at the level of the specification rather than at the level of the implementation.

One consequence of this approach is that the resulting infrastructure remains executable. Since the checkers are ordinary programs written in Agda, they can be compiled and used as practical language infrastructure while still producing proof objects internally.

Let us look at the implementation of the part of the typechecker responsible for TLam.

```
checkLambda : ∀ Γ (@0 x : Name)
                  (u : Term (a ▶ x))
                  (ty : Type a)
                  → TCM (Γ ⊢ TLam x u : ty)
checkLambda ctx x u (El s ty) = do
  let r = singScope ctx

  ⟨ y ⟩ (tu , El tvs tvt) ⟨ rtp ⟩ ← reduceToPi r ty
    "couldn't reduce a term to a pi type"

  d ← checkType (ctx , x : tu) u (El (renameTopSort r tvs) (renameTop r tvt))

  let gc = CRedR rtp (CPi CRefl CRefl)
      sp = piSort (typeSort tu) tvs

  return $ TyConv (TyLam {k = sp} d) gc
```

We can see that significantly more work is needed than to simply express the rules. First, one needs to check that ty, the type being checked against indeed reduces to a Π type. Then one checks if the body of the TLam typechecks to that reduced Π type, when the context is extended with the bound variable and the type inferred from the reduction for that variable. This is enough to provide the TyLam constructor, although one needs to use to use the proof of conversion to show that the reduced Π type corresponds to the original ty.

Chapter 3

The Guard Condition

We now investigate how the methodology described in the previous chapter can be applied to termination checking.

Rather than representing termination checking as a boolean-valued algorithm, we represent termination criteria directly as dependent types describing valid termination arguments. A termination checker then becomes a program attempting to construct inhabitants of those types.

The remainder of this chapter studies how the guard condition can be encoded within this framework, and how a checker can be made respecting the specification.

3.1 Choosing the termination criterion

As detailed in the introduction, one needs a termination criterion in order to determine if a program is terminating. An important decision is then to choose what criterion we want to use. We considered three possible approaches to verifying termination of Agda Core definitions:

- **The Size Change Principle:** This is the criterion that Agda itself uses. This makes using it for Agda Core the natural choice, as this would entail that any program normally accepted by Agda itself will be accepted by Agda Core. However, it is a pretty complex criterion, and we would like to start off with a simpler one.
- **The Guard Condition:** Another candidate the guard condition. The guard condition is used in Rocq, in which it has been modified a lot in order to accept more terminating programs [22]. However its original version is very simple and is a great candidate for a first attempt at a formal specification.
- **Only eliminators in the core syntax:** A final approach would have been to not have TCase as a term constructor in the syntax of Agda Core, and instead rely on a same method as Lean's kernel [5], which is to create recursors for each inductive type, and elaborate recursive calls to those recursors [9]. This way, termination is guaranteed if those recursors's use are well-typed, removing the need for a checker dedicated to termination checking. The main issue with this approach is that it is not clear how some of the datatypes accepted by Agda could have an eliminator elaborated from them. Another problem with this approach is that this would increase dependency on the translation from Agda to Agda Core for the creation of those eliminators, which is not verified in any way.

We decided to first go with the guard condition.

3.2 Termination Rules for the Guard Condition

As mentioned earlier, we chose the guard condition (in its simplest version) as a first candidate for checking termination in Agda Core. We will first translate it into rules encoded as indexed datatypes.

We can state the guard condition as a proposition:

A function is terminating if there exists one parameter such that all recursive calls are decreasing in it. A recursive call is decreasing in a parameter if the argument at that position in the call is a subterm of the corresponding parameter

How can we make a specification for such a proposition? We see that a parameter has to be found such that the function is called on a subterm of it. Representing that parameter is akin to a `Fin`, that is, a natural number known to be strictly smaller than some bound. We can represent it with the following type, where the scope index will be the arity of the function, such that then this type can be used to represent a pointer to one of a function's parameters.

```
data NthArg : @0 Scope Name → Set where
  ZeroNA : (@0 x : Name) → NthArg (a ▶ x)
  SucNA : (@0 x : Name) → NthArg a → NthArg (a ▶ x)
```

As discussed before, Agda Core differs from Agda in that it does not admit several clauses for each function, and instead desugars such constructs to a case tree. Here is an example of how this works:

```
countdown : ℕ → ℕ
countdown zero = zero
countdown (suc n) = countdown n
```

is roughly translated to the following Agda Core code:

```
-- countdown : ℕ → ℕ
-- countdown n = case n of
--   zero  -> zero
--   (suc m) -> countdown m
```

Now we can formalize the notion of a subterm of a parameter. The only place in an Agda Core program where a term can be observed to be a subterm of another is in a TCase, where pattern matching happens. For example, in the Agda Core pseudocode above, in the second branch of the pattern match, m is seen to be a subterm of n, since it is a parameter of the constructor making up n.

In order to check if a term is terminating, we will need to traverse it to check each of the recursive calls within it. During that traversal, we will need some sort of environment for keeping track of subterm relations introduced by case expressions. First, let's see how we can represent such a relation:

```
data SubTermRelation (@0 a : Scope Name) : Set
data SubTermRelation a where
  Unrelated : SubTermRelation a
  NonIncreasing : NameIn a → SubTermRelation a
  Decreasing : NameIn a → SubTermRelation a
```

The scope index represents all the names a variable could relate to, which we will take to be the function's arity. This is because we only care if a variable is related to a parameter of the function,

since we care only if an argument is decreasing with respect to a specific parameter. The Unrelated constructor is used for variables that have no relation to any of the parameters. Decreasing represents ones that are subterms of the variable which they are given as a parameter (which is a name in the scope, thus in the arity, thus a parameter of the function). NonIncreasing represents a variable which is of the same size as the parameter they are assigned. This is not useful for the guard condition, but is used by other condition such as the size change principle, and is thus included for later modularity.

Using that relation, we can now represent the environment that will be used during traversal:

```
data SubTermEnv : @0 Scope Name → @0 Scope Name → Set
data SubTermEnv where
  StEnvEmpty : SubTermEnv a mempty
  StEnvExtend : (@0 x : Name)
              → SubTermRelation a -- x is a subterm of this variable (if any)
              → SubTermEnv a β
              → SubTermEnv a (β ▶ x)
```

SubTermEnv associates each variable in scope to a SubTermRelation to one of the parameters of the function. The first scope SubTermEnv is indexed over represents the possible names a relation can refer to, which will be a function's arity; the second represents the names whose relations are stored in the environment, which will be the variables in use at that point in the traversal.

For convenience, we also define a record type for a function definition, containing its name (as a NameIn indexed by defScope, which is a scope defined globally which keeps track of all function names), arity, and body:

```
record FunDefinition : Set where
  no-eta-equality
  field
    index : NameIn defScope
    arity : Scope Name
    body : Term arity
```

The top-level termination proposition uses NthArg as a certificate, indicating which of the function's parameters is descending. TerminatingTerm is the type representing the traversal of the body of the function. The environment which it is initially provided (using createStEnvFromScope) simply contains each parameter set as nonincreasing (equal) with respect to themselves. That type is basically stating: If there is a parameter to the function (represented by the NthArg, and wrapped in a Maybe in case no parameter is needed to be decreasing, for example if the function does not have any recursive call), and a proof that the body of that function only contains recursive calls that are called on a subterm of that parameter, the function is terminating.

```
data Descending (@0 f : FunDefinition) : Set
data Descending f where
  DescendingIndex :
    (nthArg : Maybe (NthArg (arity f)))
    → (TerminatingTerm f nthArg
         (createStEnvFromScope (arity f))
         (body f))
    → Descending f
```

Now let us consider TerminatingTerm:

```
data TerminatingTerm (@0 f : FunDefinition)
  (@0 nthArg : Maybe (NthArg (arity f)))
  (@0 env : SubTermEnv (arity f) a) : @0 Term a → Set
```

TerminatingTerm is an inductive type and represents a proof that a given term terminates with respect to a given certificate *nthArg*. It has one constructor per constructor of Term [1] and carries around the SubtermEnvironment to be able to do the guard condition check when a recursive call happens. Here is the constructor for TerminatingTerm corresponding to Lambda abstractions.

```
TerminatingLam :
  {@0 body : Term (bind a x)}
  → TerminatingTerm f nthArg (StEnvExtend x Unrelated env) body
  ------------------------------------------------------------
  → TerminatingTerm f nthArg env (TLam x body)
```

A Lambda is terminating if its body is terminating with its bound variable added as unrelated in the Suberm environment. Now let's look at the most important constructors of TerminatingTerm, which all relate to TApp. To understand them, it is important to remember that Agda Core only has binary applications as primitive (as opposed to n-ary application), and in case a named function has more than one argument, it will be defined as a curried lambda, and applying it will result in the function of the outer TApp being itself a TApp:

```
TerminatingApp :
  {@0 function : Term a}
  {@0 argument : Term a}
  → TerminatingTerm f nthArg env function
  → TerminatingTerm f nthArg env argument
  ------------------------------------------------------------
  → TerminatingTerm f nthArg env (TApp function argument)
```

TerminatingApp is the normal inductive constructor; if both the function and the argument are found to be terminating, then the application is terminating as well. It works in combination with the two following cases.

```
TerminatingDef :
  {@0 functionName : NameIn defScope}
  → @0 Not (functionName ≡ index f)
  ------------------------------------------------------------
  → TerminatingTerm f nthArg env (TDef functionName)
```

TerminatingDef handles references to previously defined functions, and specifically does not allow recursively calling the same function: since mutual recursion is not yet supported, any reference to a different function is terminating by assumption, as it must have been checked beforehand.

[1] Except TApp, which has a case for recursive calls and one for other terminating applications

```
DecreasingNthArgApp :
  {@0 function : Term a}
  {@0 x : NameIn a}
  {@0 realNthArg : NthArg (arity f)}
  (let (func , args) = unApps function)

  → @0 nthArg ≡ (Just realNthArg)
  → @0 func ≡ TDef (index f)
  → @0 (lengthN args) ≡ indexOf (lengthScope (arity f)) realNthArg
  → @0 lookupSt env x ≡ Decreasing (getNthArg realNthArg)
  → TerminatingTermList f nthArg env args
  ------------------------------------------------------------
  → TerminatingTerm f nthArg env (TApp function (TVar x))
```

DecreasingNthArgApp is the check for recursive calls: using unApps to uncurry the application, it verifies that the certificate index is present, that the function being called is the one being defined, that the correct number of arguments appear to the left (so we are inspecting the right positional argument), and that the environment records the variable argument as a subterm of the corresponding parameter. All arguments to the left of the decreasing one must themselves be terminating.

The other most important constructor for TerminatingTerm is the one for TCase expressions:

```
TerminatingCase :
  {d : NameData}
  {@0 varName : NameIn a}
  (let iScope = dataIxScope d
       a'     = a ◀▶ iScope
       iRun = sing iScope)
  {cases : Branches a d (AllNameCon d)}
  {return : Type (a' ▶ x)}

  → TerminatingBranches f nthArg env varName
      (descend $ lookupSt env varName) cases

  ----------------------------------------------------
  → TerminatingTerm f nthArg env (TCase d iRun (TVar varName) cases return)
```

```
descend : SubTermRelation a → SubTermRelation a
descend (NonIncreasing n) = Decreasing n
descend other = other
```

```
data TerminatingBranches {a = a} {d = d} f nthArg env var rel where
  TerminatingNil : TerminatingBranches f nthArg env var rel BsNil
  TerminatingCons : ∀ {@0 c : NameCon d} {@0 cs : RScope (NameCon d)}
                      {@0 b : Branch a c} {@0 bs : Branches a d cs}
    → TerminatingBranch f nthArg env var rel b
    → TerminatingBranches f nthArg env var rel bs
    → TerminatingBranches f nthArg env var rel (BsCons b bs)
```

TerminatingCase is responsible for figuring out what relation does the variable being pattern-matched on have in the environment, use descend on that relation, which makes sure the relation becomes a subterm relation, and passes it to TerminatingBranches, which is responsible for calling TerminatingBranch on each of the branches.

```
data TerminatingBranch {a = a} {d = d} f nthArg env var rel where
  TerminatingBBranch :
    {@0 c : NameCon d}
    {rhs : Term (a ◀▶ (fieldScope c))}
    → TerminatingTerm f nthArg (updateEnv env (fieldScope c) rel) rhs
    → TerminatingBranch f nthArg env var rel (BBranch (sing c)
        ((fieldScope c) ⟨ refl ⟩) rhs)
```

TerminatingBranch expects a specification of TerminatingTerm for the clause of the branch, given an environment which is extended with all the fields of the constructor of the left-hand side of the branch, each associated with the relation being passed upstream, which is decreasing with respect to the root parameter of the function. updateEnv is just a utility function for doing that environment extension.

3.3 From Specification to Checker

With the rules in place, we can write a checker that attempts to produce a termination certificate for a given function definition.

3.3.1 Simple Brute-Force Checker

The simplest checker tries every possible certificate in turn: first no decreasing argument (Nothing as the certificate), then each positional argument one by one (Try for each possible NthArg).

```
findDescendingIndex : (f : FunDefinition) → Either String (Descending f)
findDescendingIndex f =
  catchEither
    (
      do
        descBodyNoNothArg ← checkDescendingIndex f Nothing
          (createStEnvFromScope (arity f)) (body f)
        Right $ DescendingIndex Nothing descBodyNoNothArg
    )
    (λ _ → foldr helper (Left "This function is non-terminating")
                        (iterateNthArg (arity f)))
  where
    helper : NthArg (arity f) → Either String (Descending f)
                              → Either String (Descending f)
    helper nthArg (Right res) = Right res
    helper nthArg (Left _) = do
      descBody ← (checkDescendingIndex f (Just nthArg)
                    (createStEnvFromScope (arity f)) (body f))
      Right $ DescendingIndex (Just nthArg) descBody
```

The heavy lifting is done by checkDescendingIndex, which attempts to build an object of type TerminatingTerm for a given certificate. Its most interesting case handles a potential recursive call:

```
checkDescendingIndex : (f : FunDefinition) → (nthArg : Maybe (NthArg (arity f)))
  → (env : SubTermEnv (arity f) a) → (term : Term a)
  → Either String (TerminatingTerm f nthArg env term)
```

```
checkDescendingIndex f nthArg env (TApp func (TVar x)) = do
  catchEither (checkDescendingIndexApp f nthArg env (TApp func (TVar x))) $ λ err →
    case unApps func of λ where
      (TDef fname , args) {{ eq }} → do
        realNthArg ⟨ prf2 ⟩ ← maybeToEitherWithProof nthArg
        let lengthOk = decIndex (lengthN args) (indexOf (lengthScope $ arity f) realNthArg)
            fnameOk = decNamesIn fname (index f)
            stOk    = decSubTermRelation (lookupSt env x) (Decreasing (getNthArg realNthArg))
        (True ⟨ lengthProof ⟩) ← Right lengthOk
          where _ → Left err
        (True ⟨ fnameProof ⟩) ← Right fnameOk
          where _ → Left err
        (True ⟨ stProof ⟩)    ← Right stOk
          where _ → Left "not descending"
        argsAllDescending ← checkDescendingIndexList f nthArg env args
        Right $ DecreasingNthArgApp
          prf2
          (trans (cong fst eq) (cong TDef fnameProof))
          (trans (cong lengthN (cong snd eq)) lengthProof)
          stProof
          (subst0 (TerminatingTermList f nthArg env) (sym (cong snd eq)) argsAllDescending)
      _ → Left "The function was not named"
```

The remainder of the checker follows the rules mechanically, with one clause of checkDescendingIndex for each constructor of Term.

3.3.2 Soundness but Not Completeness

An important property of this checker (and of Agda Core's approach in general) is that it is *sound* with respect to the rules but not *complete*. If the checker produces a certificate, it is guaranteed that the program respects the rules; but the checker is not guaranteed to find a certificate for every program that does respect them.

This limitation is in fact natural for a termination checker: no checker can accept all and only the terminating programs, since termination is undecidable. The checker finds a decidable subset of the terminating programs; soundness ensures that subset is correct. The limitation is best mitigated by positive tests confirming that the checker does accept known terminating functions. This approach is much lighter in terms of implementation work than trying to prove the checker complete with respect to the termination criterion.

3.3.3 Smarter Checker

Because correctness is attached to the specification rather than the search procedure, different proof search strategies can be explored without changing the underlying theory. Rather than brute-forcing over all possible NthArg candidates, we can determine a candidate in a single pass over the program by intersecting the decreasingness information from all recursive calls. Then if a candidate is found we can try to run checkDescendingIndex on it, hence removing the need ot test all possible candidates.

3.3.4 Executable Extraction

After completing those checkers, we were able to compile them to Haskell code using **Agda2hs**[8, 14]. Then, after adding it to the main runner of Agda Core (which is currently written in Haskell as well),

we were able to use it to check termination of functions. All examples explored in the Background section were judged as expected by the guard condition.

Chapter 4

The Size Change Principle

The guard condition presented in the previous chapter provides a simple and local criterion for termination. However, many terminating programs are rejected because decreases may only become apparent after several function calls have been composed together.

To address this limitation, Agda employs the Size Change Principle (SCP), originally introduced by Lee, Jones and Ben-Amram [13]. Unlike the guard condition, SCP reasons globally about how arguments evolve across chains of function calls. Rather than checking individual recursive calls in isolation, it analyses the overall call graph of a program and verifies that infinite recursion is impossible.

As with the guard condition, we would like to separate the specification of termination from the algorithm used to verify it. We therefore introduce a certificate-based formulation. The certificate takes the form of a graph containing function calls together with their associated size-change information.

4.1 Graph Certificates

Since SCP naturally supports mutually recursive definitions, termination is no longer a property of a single function but of an entire collection of functions. We therefore introduce a representation of programs.

```
data FunDefS : @0 Scope Name → Set where
  FunDefSEmpty : FunDefS mempty
  FunDefSExtend : (@0 def : Name)
    → FunDefinition
    → FunDefS a
    → FunDefS (a ▶ def)
```

```
record Program : Set where
  no-eta-equality
  field
    functions : FunDefS defScope
```

FunDefS is a list of function definitions indexed by the scope of function names. The Program record contains all function definitions belonging to the program. The helper function lookupFunc can then be used to retrieve a function definition from the program.

We next introduce graph certificates.

```
record FunctionCall (@0 p : Program) : Set where
  no-eta-equality
  field
    caller : FunDefinition
    callee : FunDefinition
    relations : SubTermEnv (arity caller) (arity callee)
```

```
record Graph (@0 p : Program) : Set where
  no-eta-equality
  field
    functionCalls : List (FunctionCall p)
```

Each FunctionCall contains a caller, a callee and a SubTermEnv describing the relation between the parameters of the caller and those of the callee.

This use of SubTermEnv differs slightly from its use in the guard condition traversal. In the guard condition, the environment tracked relations between variables currently in scope. Here, the first scope corresponds to the arity of the callee and the second scope corresponds to the arity of the caller.

Consequently, each parameter of the callee stores the relation it has to a parameter of the caller, if such a relation exists. This serves the same purpose as the size-change matrices introduced in the Background chapter. We use SubTermEnv instead because efficiency is not the primary concern of the formalization.

This representation is sufficient because subterm relations arise through pattern matching and therefore form tree-like structures rooted at the caller's parameters. A parameter of the callee can therefore be related to at most one parameter of the caller, making an environment representation a natural replacement for explicit matrices.

The graph certificate separates termination checking into two independent tasks.

First, we must verify that the graph faithfully represents the program. Every function call appearing in the program must be present in the graph certificate.

Second, we must verify that the graph satisfies the size change principle itself.

One consequence of this design is that the graph may contain edges corresponding to function calls which do not actually occur in the program. This does not compromise soundness. Additional edges can only introduce additional call paths and cycles, making the graph harder rather than easier to prove terminating. Consequently, if a graph covers all calls appearing in the program and satisfies the SCP criterion, removing superfluous edges can never invalidate the certificate.

4.2 Checking that the graph covers the program

The process of checking if the graph covers the program is very similar to how we check if all recursive calls are decreasing in the given NthArg in the rules for the guard condition. It is again a traversal, and the only way in which it differs is that, when a call happens, it has to check whether the call with its inferred call matrix is present in the certificate graph.

```
data GraphTerm {@0 p : Program} (@0 g : Graph p) (@0 f : FunDefinition)
                 (@0 env : SubTermEnv (arity f) a) : @0 Term a → Set
```

GraphTerm is the datatype of the rule stating that a given term is fully covered by its index Graph.

```
GraphAppCall :
  {@0 body : Term a}
  {@0 callee : FunDefinition}
  (let (func , args) = unApps body)
  → @0 func ≡ TDef (index callee)
  → @0 graphContainsCall (functionCalls g) f callee env args ≡ True
  → GraphTerm g f env body
```

GraphAppCall is a constructor of GraphTerm that stands for the rule that states that a function call is covered by the certificate Graph.

```
graphContainsCall : {@0 p : Program} → List (FunctionCall p) → (caller : FunDefinition)
                    → FunDefinition → SubTermEnv (arity caller) ξ → List (Term ξ) → Bool
graphContainsCall [] caller' callee' subtermEnv args = False
graphContainsCall {ξ} (call :: calls) caller' callee' subtermEnv args =
  case (decNamesIn (index caller') (index $ caller call)) of λ where
    (False ⟨ _ ⟩) → graphContainsCall calls caller' callee' subtermEnv args
    (True ⟨ pp ⟩) → case (decNamesIn (index callee') (index $ callee call)) of λ where
      (False ⟨ _ ⟩) → graphContainsCall calls caller' callee' subtermEnv args
      (True ⟨ _ ⟩) →
        case (compareEnvironments (relations call) (subst0 (λ sc → SubTermEnv sc ξ)
          (axiom1 caller' (caller call) pp) subtermEnv) args) of λ where
          False → graphContainsCall calls caller' callee' subtermEnv args
          True → True
```

graphContainsCall is used by GraphAppCall and ensures the index graph contains the given function call. It is responsible for iterating through the list of FunctionCall in the certificate Graph and, for each of those, checking that the callers and callees are the same, and that the SubTermEnvironement are the same.

```
compareEnvironments : SubTermEnv a β → SubTermEnv a ξ → List (Term ξ) → Bool
compareEnvironments StEnvEmpty _ [] = True
compareEnvironments (StEnvExtend x rel rest) env (TVar name :: args) =
  case (decSubTermRelation (lookupSt env name) rel) of λ where
    (True ⟨ _ ⟩) → compareEnvironments rest env args
    (False ⟨ _ ⟩) → False
compareEnvironments (StEnvExtend x rel rest) env (arg :: args) =
  case rel of λ where
    Unrelated → compareEnvironments rest env args
    _ → False
compareEnvironments _ _ _ = False
```

compareEnvironments is used by graphContainsCall, and is the most complex check that is done in it. It checks that, in the environment accumulated at that point in the term traversal, the arguments in the function call have the same SubTermRelation as the ones registered in the call of the graph currently being compared.

4.3 An Abstract Formulation of SCP

The original presentation of SCP is fundamentally a statement about infinite executions. Informally, it states that every infinite sequence of function calls must contain a parameter which decreases infinitely often.

The original paper develops this perspective using Büchi automata. Infinite call sequences are viewed as infinite words accepted by an automaton derived from the call graph. Termination then follows from showing that every accepted infinite execution encounters infinitely many decreasing transitions.

To capture this viewpoint directly, we introduce a coinductive representation of infinite call sequences.

```
record InfiniteCallChain {@0 p : Program} (@0 g : Graph p)
         (@0 start : FunDefinition) : Set where
  coinductive
  field
    @0 next : FunDefinition
    headChain  : FunctionCall p
    @0 inGraph : ListIn (λ x → x .caller .index == headChain .caller .index
          && x .callee .index == headChain .callee .index) (g .functionCalls)
    @0 headCallerEq : caller headChain ≡ start
    @0 headCalleeEq : callee headChain ≡ next
    tailChain    : InfiniteCallChain g next
```

An InfiniteCallChain represents an infinite sequence of function calls in the graph. Each element consists of a current function call together with another infinite call chain beginning at the callee.

Reasoning about SCP requires considering finite portions of these infinite chains.

```
data CallChainPrefix {@0 p : Program} (@0 g : Graph p)
       : (@0 start end : FunDefinition) → @0 InfiniteCallChain g start → Set where
  PrefixSing : {@0 start : FunDefinition} (chain : InfiniteCallChain g start)
    → CallChainPrefix g start (chain .InfiniteCallChain.next) chain
  PrefixCons : {@0 start end : FunDefinition} {chain : InfiniteCallChain g start}
    → CallChainPrefix g (chain .InfiniteCallChain.next)
        end (chain .InfiniteCallChain.tailChain)
    → CallChainPrefix g start end chain
```

A CallChainPrefix represents a finite prefix of an infinite call sequence. Since decreases may only become visible after composing multiple calls, SCP does not require every individual step to decrease.

Instead, we compute the cumulative relation induced by an entire prefix.

```
computePrefixSubTermRelations : {@0 p : Program} {@0 g : Graph p}
                                {@0 start end : FunDefinition}
                                {@0 chain : InfiniteCallChain g start}
  → CallChainPrefix g start end chain → SubTermEnv (arity start) (arity end)
computePrefixSubTermRelations (PrefixSing chain) =
  subst0 (λ s → SubTermEnv s (arity (chain .next)))
    (cong arity (headCallerEq chain))
    (subst0 (λ e → SubTermEnv (arity (caller (chain .headChain))) (arity e))
      (headCalleeEq chain)
      (relations (chain .headChain)))
computePrefixSubTermRelations (PrefixCons {chain = chain} rest) =
  computeTransitiveSubTermRelations
    (subst0 (λ s → SubTermEnv s (arity (chain .next)))
      (cong arity (headCallerEq chain))
      (subst0 (λ e → SubTermEnv (arity (caller (chain .headChain))) (arity e))
        (headCalleeEq chain)
        (relations (chain .headChain))))
    (computePrefixSubTermRelations rest)
```

computePrefixSubTermRelations composes the relations associated with every call in a prefix. Conceptually, this corresponds to composing size-change matrices in the standard presentation of SCP.

Once a prefix has been consumed, we continue with the remaining suffix.

```
splitPrefix : {@0 p : Program} {@0 g : Graph p} {@0 start end : FunDefinition}
              {@0 chain : InfiniteCallChain g start}
            → CallChainPrefix g start end chain
            → InfiniteCallChain g end
splitPrefix (PrefixSing chain) = tailChain chain
splitPrefix (PrefixCons rest) = splitPrefix rest
```

splitPrefix removes a finite prefix from an infinite call chain and returns the remaining infinite suffix.

Using these components, we can formulate the central notion of infinite descent.

```
record InfinitelyDescending {@0 p : Program} (@0 g : Graph p) {@0 start : FunDefinition}
                            (@0 var : NameIn (arity start))
                            (@0 chain : InfiniteCallChain g start) : Set where
  coinductive
  field
    @0 mid : FunDefinition
    @0 var' : NameIn (arity mid)
    prefix      : CallChainPrefix g start mid chain
    @0 descends : lookupSt (computePrefixSubTermRelations prefix) var' ≡ Decreasing var
    restChain   : InfinitelyDescending g var' (splitPrefix prefix)
```

InfinitelyDescending is a coinductive predicate stating that a parameter decreases infinitely often along an infinite call sequence.

Rather than requiring a decrease at every step, the definition allows an arbitrary finite prefix to be chosen. The composed relation associated with that prefix must witness a strict decrease. Crucially,

the descent must follow a consistent thread through the arguments. If the composed relation shows that a parameter *var* at the start of the prefix is related to a parameter *var'* at the end of the prefix, then the coinductive proof continues from *var'* in the remaining suffix. In other words, we do not repeatedly choose unrelated parameters; we track how a single argument position evolves through successive calls.

This corresponds closely to the mathematical statement of SCP. Along every infinite execution, one must always be able to find another future decrease while following such a thread through the call sequence.

The resulting termination criterion is then:

```
record Terminating (@0 p : Program) (@0 g : Graph p) : Set where
  no-eta-equality
  field
    @0 allInfiniteChainsDescend :
      (start : FunDefinition) (chain : InfiniteCallChain g start)
      → Σ0 (NatIn (arity start)) (λ var → InfinitelyDescending g var chain)
    @0 graphCoversCalls : GraphCoversCalls g (functions p)
```

The first field states that every infinite call chain contains an infinitely descending parameter. The second field requires that the graph certificate covers all calls appearing in the program.

This formulation is very close to the mathematical statement of SCP, but it is also highly abstract. It directly quantifies over infinite executions and therefore provides little guidance for constructing an executable checker.

4.4 An Operational Formulation Using Idempotent Cycles

To obtain a more practical characterization, we turn to a second formulation based on finite graph structures.

The SCP paper shows that it is sufficient to reason about particular cycles in the transitive closure of the call graph. More specifically, it suffices to consider idempotent cycles, that is, cycles whose associated relation remains unchanged when composed with itself.

This leads to a significantly more operational formulation.

```
data CallChain {@0 p : Program} (@0 g : Graph p) : @0 FunDefinition
       → @0 FunDefinition → Set where
  ChainSing : (fc : FunctionCall p)
            → @0 ListIn (λ x → x .caller .index == fc .caller .index
                 && x .callee .index == fc .callee .index) (g .functionCalls)
            → CallChain g (caller fc) (callee fc)
  ChainCons : {@0 end : FunDefinition}
            → (fc : FunctionCall p)
            → @0 ListIn (λ x → x .caller .index == fc .caller .index
                 && x .callee .index == fc .callee .index) (g .functionCalls)
            → CallChain g (callee fc) end
            → CallChain g (caller fc) end
```

A CallChain represents a finite sequence of calls in the graph.

```
record Cycle (@0 p : Program) (@0 g : Graph p) : Set where
  no-eta-equality
  constructor MkCycle
  field
    @0 {func} : FunDefinition
    chain : CallChain g func func
```

A Cycle consists of a call chain whose starting and ending function coincide.
As before, we can compose the relations associated with all calls in the chain.

```
computeChainsSubTermRelations : {p : Program} {g : Graph p} {caller callee : FunDefinition}
  → CallChain g caller callee → SubTermEnv (arity caller) (arity callee)
computeChainsSubTermRelations (ChainSing fc _) = relations fc
computeChainsSubTermRelations (ChainCons fc _ chain) =
  computeTransitiveSubTermRelations (relations fc) (computeChainsSubTermRelations chain)
```

computeChainsSubTermRelations computes the cumulative size-change relation associated with a cycle.

We can then define what it means for a cycle to witness progress.

```
data DescendingRecursiveCall : @0 SubTermEnv a β → @0 β ⊆ a → Set where
  DescendingRecursiveCallMatch :
    {@0 nameIn : NameIn a}
    {env : SubTermEnv a β}
    (let ⟨ name ⟩ p = nameIn)
    {prf : bind β name ⊆ a}
    → DescendingRecursiveCall (StEnvExtend name (Decreasing nameIn) env) prf
  DescendingRecursiveCallExtend :
    {env : SubTermEnv a β}
    {name : Name}
    {prf : (bind β name) ⊆ a}
    {rel : SubTermRelation a}
    → DescendingRecursiveCall env (subWeaken' prf)
    → DescendingRecursiveCall (StEnvExtend name rel env) prf
```

DescendingRecursiveCall states that some parameter decreases with respect to itself in the composed relation.

```
record TerminatingCycle {@0 p : Program} (@0 g : Graph p) (@0 c : Cycle p g) : Set where
  no-eta-equality
  field
    descendingParameter : DescendingRecursiveCall (computeChainsSubTermRelations (chain c))
                                                  subRefl
```

A cycle is terminating whenever its composed relation contains such a decrease.
The crucial observation is that not every cycle needs to be examined individually.

```
data IdempotentCycle {@0 p : Program} (@0 g : Graph p) : Set where
  MkIdempotentCycle :
    (c : Cycle p g)
    → @0 computeChainsSubTermRelations (chain c) ≡
         computeTransitiveSubTermRelations
           (computeChainsSubTermRelations (chain c))
           (computeChainsSubTermRelations (chain c))
    → IdempotentCycle g
```

An IdempotentCycle is a cycle whose composed relation is equal to its composition with itself.

Intuitively, such cycles represent stable behaviours of the graph. Repeating the cycle cannot produce any new information.

The resulting termination criterion becomes:

```
record Terminating (@0 p : Program) (@0 g : Graph p) : Set where
  no-eta-equality
  field
    @0 allIdempotentCyclesDescend :
      (ic : IdempotentCycle g) → TerminatingCycle g (idempotentCycleToCycle ic)
    @0 graphCoversCalls : GraphCoversCalls g (functions p)
```

A graph is terminating if every idempotent cycle is terminating and if the graph covers all calls appearing in the program.

Compared to the previous formulation, this criterion is much closer to the structure of practical SCP implementations.

4.5 Writing the checker

While the specification itself can be expressed relatively naturally, deriving an executable certified checker for it introduces several additional difficulties.

4.5.1 Agda Core does not yet support mutually recursive functions

We added Program as a record type containing a list of FunDef for the termination rules. However, such a construct is not present in Agda Core at the highest level, and for typechecking, where definitions are instead processed separately in order without a bigger structure containing them. In order to make a runnable checker which Agda Core can use, we will first need to extend Agda Core to support mutually recursive functions, and that includes add typechecking for that feature as well.

4.6 Relating the Two Formulations

The two formulations presented above describe the same intuition from different perspectives.

The abstract formulation reasons directly about infinite executions and infinite descent. The operational formulation reasons about finite cycles and decreasing self-loops.

A natural correctness theorem would state that the operational formulation implies the abstract one. Such a result would justify using the more practical cycle-based criterion while preserving the conceptual meaning provided by the infinite-execution formulation.

Interestingly, the connection between these viewpoints is non-trivial. The original SCP paper establishes the equivalence using a Ramsey-theoretic argument. Ramsey's theorem allows one to extract regular behaviour from an arbitrary infinite call sequence. Intuitively, if an infinite execution exists, some pattern of composed relations must occur infinitely often. This repeated behaviour can then be shown to correspond to an idempotent cycle.

Thus, the operational criterion is not merely a convenient approximation. It arises from a combinatorial property connecting infinite executions to finitely representable cyclic behaviour. Proving the correspondence of the two will not be an easy task.

4.6.1 The rules expect all possible cycles

In the more operational variant of the rules, a central difficulty remains. The specification quantifies over all *idempotent cycles*. Although this already restricts attention to stabilized behaviour, the resulting collection is still infinite in general, since repeated composition and closure can generate infinitely many distinct idempotent representatives.

Consequently, a checker cannot hope to enumerate all idempotent cycles directly. Instead, non-certified implementations of size-change termination construct a finite closure of size-change relations. For a given function arity, only finitely many call matrices exist, and repeated composition eventually stabilises. This yields a finite saturated set closed under composition, which serves as a finite generator for all idempotent behaviour.

From the perspective of a certified checker, however, this introduces a fundamental design choice. One possibility is to complexify the rules so that they explicitly quantify only over a curated subset of idempotent cycles derived from this finite closure. While operationally convenient, this would make the specification less abstract and harder to read, which is undesirable in systems such as Agda Core.

The alternative is to keep the original rules unchanged, and instead prove that the curated finite subset is sufficient: every idempotent cycle required by the specification is represented by some element of the finite closure. This shifts the burden from the specification to a meta-theoretic completeness proof.

Thus, although the rule is stated over all idempotent cycles, implementations never manipulate this infinite collection directly. Instead, they rely on a finite saturated structure, together with a proof that it adequately covers the full class of idempotent cycles. This illustrates a general tension between elegant declarative specifications and the additional structure required for certified executable algorithms.

Chapter 5

Related Work

5.1 MetaRocq

As mentioned in the introduction, the work that is closest to Agda Core is MetaRocq [20, 19]. It consists of a certification of the core language of Rocq in Rocq, just like Agda Core is for Agda. It is however much larger in scale. While Agda Core only aims to declare rules as a certification and derive checkers that respect it, MetaRocq also has as an objective to reason about the metatheory of the language.

Concerning termination checking, MetaRocq axiomatizes the guard condition from Rocq [21]. The guard condition in Rocq is much more complex than the one we use, having been worked on by many different authors, so making a formal certification for it is no easy task. The way Metarocq deals with that is that it relies on the Rocq implementation of it for termination checking, and it axiomatizes its properties, such as stability under one-step reduction and stability under parallel substitution, for use in the proofs of the metatheory of the language.

In a more recent work [22], the author went about implementing that guard condition in MetaRocq, so coded in Rocq. However they did not declare a termination specification and a checker respecting it, opting instead to just have the checker, which means, in that version, the guard condition remains part of the TCB. The author cites as a future work to formally specify the guard condition, to prove that it is sound (implies termination), and to create a checker. Our work formally specifies a simpler version of the guard condition, and implements a checker for it, however the proof of soundness is out of scope, as Agda Core simply aims to specify the theory without proving its correctness.

5.2 Lean4Lean

Another closely related work is that of Lean4Lean [6]. Lean4Lean is a formalization of the core language of Lean [10], just like Agda Core is for Agda. However, the theory and checkers are much more separate than in Agda Core. The checkers are more standard and do not produce an object with the typing rules as types, and the theory is defined completely independently. Then proofs are provided that if the typechecker validates an expression, it implies the separate theory also recognizes the expression as well-typed. They call this "external checkers". Lean does not handle termination checking the same way Rocq or Agda does. It does not have case expressions as a core constructs, and instead the elaborator[9] (which is responsible for turning syntax into core syntax) creates recursors for each inductive type [5], which are responsible for any possible recursion, which is what is responsible for non termination in Agda or Rocq. The elaborator is not part of the trusted core, and thus is not meant to be verified, and any produced recursor which typechecks is guaranteed to terminate, hence removing the need for a termination checker altogether

5.3 Sized Types

An relatively new approach to termination checking is the use of sized types [1, 18]. Rather than implementing termination checking in a separate checker based on some criterion, sized types integrate termination checking directly into type checking.

The central idea is to annotate inductive types with size indices representing upper bounds on the depth of values. Recursive functions are then typed in such a way that recursive calls must be performed on arguments of strictly smaller size. Termination checking is therefore contained in typechecking, meaning that if a function typechecks, it has to be terminating.

As an example, one may consider a sized version of natural numbers `Nat` i, where the size index i bounds the height of the value. The successor constructor would increase the size bound, while recursive calls would be required to operate on strictly smaller indices. This allows the type system itself to enforce well-founded recursion.

Sized types are more expressive than simple structural criteria such as the guard condition. In particular, they can naturally handle many programs which require reasoning about more subtle decreases than direct subterm relations. They also avoid some of the operational complexity of termination checkers based on graph exploration, since termination arguments are embedded directly into typing derivations.

However, integrating termination into the type system also increases the complexity of the theory and implementation. In practice, sized types require additional machinery for size polymorphism, subtyping, and constraint solving over size expressions. Moreover, while Agda includes support for sized types, the feature remains experimental and there are known inconsistencies and implementation issues [3]. Sized types also require much more work for the programmer, as they need to annotate functions with size information. Some work has been done to automate this process: Nisht [15] proposed a size inference algorithm that can automatically reconstruct size annotations for terms in a System F_ω-like fragment, proving its soundness and establishing linear time complexity in the size of the syntax. An implementation extending Agda with this algorithm was submitted as a pull request to the main codebase [16], but remains unmerged, and full inference in the dependently typed setting remains an open challenge.

From the perspective of Agda Core, sized types represent a significantly different design philosophy from the one explored in this thesis. Adding those constructs as part of Agda Core goes against the principle of a core language, which is meant to only contain simple constructs. Our work keeps termination checking separate from ordinary typing and instead studies how termination criteria themselves can be represented as proof-relevant specifications together with certified proof-search procedures. Moreover, this technique does not remove the need of a separate termination checker entirely, as size parameters still need to be decreasing in each recursive calls. Nevertheless, sized types remain an important direction for future work, particularly because they suggest the possibility of integrating richer termination reasoning directly into the certified typechecking infrastructure itself.

5.4 Termination Checking in the $\lambda\Pi$-Calculus Modulo Theory

Genestier's work on termination checking [11] is somewhat related to our own work. The author implements size change termination for the $\lambda\Pi$-calculus modulo theory. The $\lambda\Pi$-calculus modulo theory extends dependent type theory with user-defined rewrite rules. Since unrestricted rewriting can break normalization, termination checking becomes necessary in order to preserve the consistency of the system. In that setting, the size change principle is used as a tool for reasoning about the normalization behaviour of rewrite systems and recursive definitions.

The goal of that work is different from our own. They aim to prove semantic normalization of their calculus, using the size change termination criterion, which they independently implemented. Our work instead studies how the size change principle itself can be represented inside a proof-relevant framework. We assume the correctness of the criterion and investigate how to express its rules as

dependent types together with certified checking infrastructure respecting those rules.

This difference reflects a broader distinction in objectives. Genestier's work is primarily concerned with metatheoretical properties of the calculus itself, while our work focuses on the relationship between declarative specifications and executable certified infrastructure. In particular, rather than proving normalization from the size change principle, we investigate the difficulties involved in deriving proof-producing checkers for the criterion itself.

Chapter 6

Conclusion

In this thesis, we investigated how termination checking can be integrated into the certified infrastructure of Agda Core. We studied how termination criteria can be represented constructively inside the theory itself, and how executable checkers can be derived from those specifications.

We first formalized a simplified version of the guard condition for Agda Core. The criterion could be expressed relatively naturally using indexed datatypes representing decreasing recursive calls together with environments tracking subterm relations introduced by pattern matching. From this specification, we derived executable checkers capable of constructing explicit termination certificates, and successfully extracted them to runnable Haskell code integrated into Agda Core.

We then investigated the size change principle, a substantially more expressive criterion capable of handling mutually recursive programs and more complex termination arguments. While its declarative specification could also be represented naturally using graph certificates and call relations, deriving an executable certified checker exposed additional constructive difficulties. In particular, reasoning about infinitely many possible call cycles requires either enriching the specification with finite completeness arguments or proving separately that a finite subset of cycles is sufficient.

This highlights an important distinction between declarative termination specifications and the executable procedures used to check them. Structural criteria such as the guard condition admit relatively direct implementations from their abstract rules, while more global criteria such as the size change principle expose a wider gap between the specification and the checker. In particular, the rules of the size change principle are most naturally expressed in a highly abstract form quantifying over arbitrary call cycles, whereas executable checking procedures rely on finite operational arguments that are not explicit in the specification itself. Bridging this gap therefore requires substantially more additional structure than in ordinary implementations, where such operational reasoning can remain implicit.

Several directions remain for future work. Supporting mutually recursive definitions in Agda Core would be necessary for a fully executable implementation of the size change principle. More importantly, one would need to resolve the problem of representing and reasoning about infinitely many cycles in a way compatible with executable certified checking.

Overall, this work demonstrates that termination checking can be integrated into Agda Core's methodology, and that representing termination criteria directly inside the implementation language provides a useful framework for studying the relationship between declarative specifications and executable certified infrastructure.

Broader Impact

This thesis contributes to the broader effort of increasing trust in proof assistants and formal verification systems. Modern proof assistants such as Agda, Rocq, and Lean are increasingly used for the verification of software, programming language implementations, and mathematical results. As these

systems are applied to more critical domains, confidence in the correctness of their internal components becomes increasingly important.

The work presented in this thesis explores a methodology in which correctness properties are specified directly as dependent types, while executable checkers are implemented as proof search procedures for those specifications. By applying this approach to termination checking in Agda Core, the project demonstrates how important metatheoretical properties can be expressed independently of the algorithms used to verify them. This separation can improve maintainability, facilitate experimentation with different checking strategies, and provide stronger guarantees about the correctness of implementations.

A particular contribution of this work is the exploration of certificate-based formulations of termination criteria. Such formulations make explicit what evidence is required for a program to be accepted, providing a foundation for future work on formally verified implementations of termination checking. More broadly, this aligns with ongoing trends in computer science toward increased transparency, reproducibility, and machine-checked correctness of software systems.

The practical impact of the work is currently limited by the scope of Agda Core itself. The guard condition implementation is executable and can be integrated into the current system, while the formulation of the Size Change Principle remains largely a specification-level contribution that requires additional infrastructure before it can be deployed. Consequently, the immediate beneficiaries of this work are researchers and developers working on proof assistants and formally verified programming languages rather than end users of software systems.

There are also limitations to the approach. Formal specifications often introduce additional complexity and implementation effort compared to conventional algorithms. Furthermore, a formally verified checker is only one component of a larger trusted computing base; the overall reliability of a proof assistant still depends on the correctness of many other components. As a result, the methods explored in this thesis should be viewed as one step toward greater assurance rather than a complete solution.

Nevertheless, improving the trustworthiness of proof assistant infrastructure has the potential to produce long-term benefits. As formally verified software becomes more widely adopted, advances in the correctness and transparency of the tools used to construct such software can contribute indirectly to the reliability of systems on which researchers, developers, organizations, and society increasingly depend.

Bibliography

[1] Andreas Abel. MiniAgda: Integrating Sized and Dependent Types. *Electronic Proceedings in Theoretical Computer Science*, 43:14–28, December 2010. arXiv:1012.4896 [cs]. URL: http://arxiv.org/abs/1012.4896, doi:10.4204/EPTCS.43.2.

[2] Andreas Abel. Irrelevance in Type Theory with a Heterogeneous Equality Judgement. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, pages 57–71, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-19805-2_5.

[3] Agda Developers. Open issues labeled "sized types" in agda. https://github.com/agda/agda/issues?q=is%3Aissue%20state%3Aopen%20label%3Afalse%20label%3A%22sized%20types%22, 2026. GitHub issue tracker, accessed 2026-05-13.

[4] N G De Bruijn. LAMBDA CALCULUS NOTATION WITH NAMELESS DUMMIES, A TOOL FOR AUTOMATIC FORMULA MANIPULATION, WITH APPLICATION TO THE CHURCH-ROSSER THEOREM.

[5] Mario Carneiro. The type theory of Lean. Master's thesis, Carnegie Mellon University, April 2019. Master's thesis. URL: https://github.com/digama0/lean-type-theory/releases/tag/v1.0.

[6] Mario Carneiro. Lean4Lean: Verifying a Typechecker for Lean, in Lean, September 2025. arXiv:2403.14064 [cs]. URL: http://arxiv.org/abs/2403.14064, doi:10.48550/arXiv.2403.14064.

[7] Jesper Cockx. Dependent pattern matching and proof-relevant unification.

[8] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 108–122, Ljubljana Slovenia, September 2022. ACM. URL: https://dl.acm.org/doi/10.1145/3546189.3549920, doi:10.1145/3546189.3549920.

[9] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in Dependent Type Theory.

[10] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language (System Description).

[11] Guillaume Genestier. Termination checking in the λΠ-calculus modulo theory.

[12] Eduarde Giménez. Codifying guarded definitions with recursive schemes. volume 996, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. Book Title: Types for Proofs and Programs Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/3-540-60579-7_3, doi:10.1007/3-540-60579-7_3.

[13] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 81–92, New York, NY, USA, January 2001. Association for Computing Machinery. doi:10.1145/360204.360210.

[14] Bohdan Liesnikov and Jesper Cockx. Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language. In Oleg Kiselyov, editor, *Programming Languages and Systems*, pages 63–83, Singapore, 2025. Springer Nature. doi:10.1007/978-981-97-8943-6_4.

[15] Kanstantsin Nisht. Type-Based Termination Checking in Agda.

[16] Kanstantsin Nisht. Add type-based termination checker (pull request #7152). GitHub pull request, agda/agda, 2024. Unmerged. Companion to master's thesis [**?**]. URL: https://github.com/agda/agda/pull/7152.

[17] Ulf Norell. Towards a practical programming language based on dependent type theory.

[18] Jorge Luis Sacchini. On type-based termination and dependent pattern matching in the calculus of inductive constructions.

[19] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. URL: https://inria.hal.science/hal-02167423, doi:10.1007/s10817-019-09540-0.

[20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Snapshot of MetaCoq associated to the article*, 4(POPL):8:1–8:28, December 2019. URL: https://dl.acm.org/doi/10.1145/3371076, doi:10.1145/3371076.

[21] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq , in Coq. *Journal of the ACM*, 72(1):1–74, February 2025. URL: https://dl.acm.org/doi/10.1145/3706056, doi:10.1145/3706056.

[22] Yee-Jian Tan and Yannick Forster. Towards Formalising the Guard Checker of Coq. Technical report, Ecole polytechnique ; Inria - Paris, March 2025. URL: https://inria.hal.science/hal-04983786.

[23] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936. doi:10.1112/plms/s2-42.1.230.