Dependent Types and Conversion Checking

A literature survey on implementation techniques for type systems

Maria Khakimova[1]

Supervisor(s): Jesper Cockx[1]**, Bohdan Liesnikov**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

While dependent types can allow programmers to verify properties of their programs, implementing a type checker for a dependent type theory is often difficult. This is due to the fact that, in the presence of dependent types, deciding the equality of types - conversion checking - becomes non-trivial. Due to an identified gap in literature regarding a survey of existing techniques for implementing conversion checking in the presence of dependent types, this paper aims to provide an exploratory overview of the current state of this field.

We identify five distinct implementation strategies within this paper. Four of these techniques were different from a theoretical standpoint - Naïve Substitution, Normalisation by Evaluation, Equality through "Shape" Analysis, Normalisation by Hereditary Substitution, and a technique using congruence closure. They all have different benefits and drawbacks regarding their portability, extendability to richer type systems, efficiency, and decidability. Additionally, three techniques that focused on improving the efficiency of conversion checking through the use of an abstract machine or compilation into native code were found.

1 Introduction

In type theory, a dependent type depends on some terms or elements of other types [1; 2], and their presence in a type theory can thus allow for more expressive types. This can result in more errors being caught by the compiler [3], which is beneficial for programmers working with a language implementing the dependent type system. Additionally, dependent types allow for program verification as they add precision to types [4, p. 1]. For example, proofs can be expressed as a dependent program through the Curry-Howard correspondence, which permits for any logical statements to be expressed as a dependant type [2]. However, the implementation of dependent type systems is often more complex than for simpler type systems.

One of the main reasons for the difficulties of implementing a dependent type theory is that a dependent type can contain a piece of code [5]. This makes conversion checking (a fundamental component of a typed functional programming implementation [6]) difficult, as deciding the equality of types now also has to decide the equality of terms.

It is important for a developer of a new dependently typed programming language to consider which conversion checking algorithm to use. This is because different algorithms support different functionalities and type theories. Additionally, it is also important to consider what they want from the type equality. For example, while a flexible equality of terms will ease the burden of proof, it can also make the type checking undecidable. These considerations can influence a developer's choice of the algorithm.

This survey will thus provide an exploratory overview of the different implementation methods for the conversion checking of dependent types. With the current state of literature, it is currently difficult to locate and compare different algorithms, which is compounded by the inconsistencies in terminology used. As there is currently no literature survey that covers and compares existing implementation methods, this paper will aim to cover that gap in research. However, it is still important to note that this is an exploratory survey, and therefore will likely not be able to cover all existing techniques.

The research question that this literature survey will thus focus on answering will be:

What different implementation techniques for conversion checking of dependent types have been proposed in the literature?

As this question is relatively broad, it is split into further sub-questions to aid in structuring the research:

- *What are the advantages and disadvantages of different implementation techniques*
- *Under what circumstances are certain existing implementation techniques recommended over others?*

The paper will be structured as follows. Section 2 will discuss the methodology used in the exploratory literature survey, and section 3 will go over the measures this paper will take to ensure that the research is responsible. Then, section 4 will provide some brief definitions of concepts that may be unfamiliar to non-experts in this field, before section 5 provides an overview of all the identified techniques. Section 6 will then offer a discussion on the identified techniques with advice to implementers, as well as the limitations faced by this survey. Finally section 7 will conclude the paper, and offer some insight into potential future work.

2 Methodology or Problem Description

This section covers the methodology used within this literature survey. This will include the method of collecting the information, as well as details on how technique classification and comparison will be done.

2.1 Collection of Information

Initially, collection will be done of research papers that discuss and present implementation techniques for conversion checking algorithms. This will be done using several techniques, one of which is a keyword search. With this method, queries will be formulated based on identified keywords and their synonyms, and used to search within article databases. Additionally, the reference list and bibliography of relevant papers will be studied to identify papers and techniques that have been built upon. Finally, Connected Papers[1] will be used to identify derivative papers, as well as papers with overlapping citations. This will allow for more recent papers to be found, ensuring that more modern (and perhaps less popular) techniques will also be covered.

However, due to both the current state of the literature and the nature of the topic being implementation techniques, it

[1]www.connectedpapers.com

is possible that not all approaches will be covered by peer-reviewed articles. Due to this, the search will have to be expanded to blog posts and podcasts by reputable authors in the field. Additionally, documentation and git repositories for languages implementing dependent types will be examined. This is because README files, git issues, git commit messages, and comments in code can provide insight into the implementation techniques, as well as their advantages and disadvantages. While these sources may be considered to be less reliable than a peer-reviewed article, their consideration will allow greater representation of primary sources, and may be necessary for a comprehensive review.

2.2 Technique Identification

The implementation techniques will be clustered for easier discussion. This is due to multiple reasons, one of which is that some papers do not include the name of the technique that they are implementing. Sometimes this is because it is a novel technique, although it can also happen for pre-existing named techniques. On the other hand, the naming of different implementation techniques can be inconsistent, with different names being used for the same method. Finally, some techniques are small variations on others, and the differences may either be interesting to catalogue and discuss or be minor enough to be ignored. Manual clustering of the papers will therefore allow for better structure within the paper.

2.3 Technique Comparison

Ultimately, in order to give advice to implementers, a comparison of the identified techniques will be done. To do this, the methods will be compared based on the simplicity of the implementation, as well as its performance. However, these judgements will often be subjective, as this paper is a literature survey that will not be carrying out experiments. Additionally, comparisons will be made based on the extendibility of the techniques to other and/or richer type systems, and the portability of the algorithms between type theories. Advice will also be given based on the supported language features of different algorithms.

3 Responsible Research

While the ethics of a literature survey in type theory may be simpler than in other fields, it is still important to consider the potential impact of this paper. For this reason, this section will cover the efforts that this paper has made to ensure that the research is ”responsible”.

Specific care will be taken to not misrepresent the current state of the field. While this paper aims to provide an exploratory overview, it will also try to avoid biases towards certain techniques. To achieve this, primary sources will be used as much as possible, but implementers' feedback on the techniques will also be taken into account. Additionally, part of the data will be collected through a broad keyword search as mentioned in 2.1. However, it is important to note that the snowball technique was one of the main data collection techniques that was used. This may mean that the papers collected may have some biases, as they will be strongly linked to each other. This can also result in some techniques being missed, which can also lead to a misrepresentation of the current state of literature. Overall, while efforts have been made to ensure that the paper is as representative as possible, it is important to note that this literature survey has some limitations in this regard.

Steps have also been taken to ensure the reproducibility of this research. For this purpose, the methodology has been outlined in detail in section 2, and citations have been used generously to ensure that it is clear where specific claims come from. Page numbers are used in citations for longer publications like books and PhD theses for easier reproducibility.

4 Definitions

Before detailing the different identified implementations, it is important that all readers have an understanding of the terminology used. This is because the aim of this paper is to provide an exploratory overview of the topic to all audiences, including those less familiar with type theory and conversion checking. Thus, this section will go over some of the terminology that may be new to readers who are not experts in this field.

To the extent of our knowledge, in dependent type theory convertibility is analogous to definitional equality, and both terms will be used within the paper. However, it is also important to note that often, when referring to dependent type theories, the terms *definitional equality* and *judgemental equality* are used interchangeably. For the purpose of clarity, this paper will avoid using the term *judgemental equality*, as there can be cases where the two differ in dependent type theory.

For conversion checking, it is important for the implementer to know which equivalence relations they want the language to support.

The simplest equivalence relation is α-equivalence. Two terms being α-equivalent means that the only differences between them are in the names of bound variables [7; 8]. As an example, $\lambda x.\, x\, y$ and $\lambda z.\, z\, y$ are α-equivalent. α-equivalence can often be equivalent to syntactic equivalence, especially when the type theory is formulated with devices such as De Bruijn indices [9].

β-equivalence is also an equivalence relation that many languages and type-checkers support. For two terms to be β-equivalent, they must be able to be β-reduced into the same β normal form [10]. To perform β-reduction, the λ is removed from the function, and the argument is substituted for the function's parameter in the body [11].

The last equivalence relation that will be covered is η-equivalence, which is said to express the idea of extensionality [12]. Therefore, two functions are seen as equal when they give identical results for all arguments. Unlike β-conversion, both η-reduction and -expansion may be necessary to give an η normal form [13], although the usage of both is often instead for purposes of efficiency.

Whilst these are not the only equivalence relations that may be considered by a language[2], this paper will mostly focus

[2]As an example, Coq considers terms to be convertible when they are $\beta\delta\iota\zeta\eta$-equal[14]

on conversion checkers that allow for the satisfaction of $\beta\eta$-equality.

5 Implementation Techniques

A variety of techniques can be used to implement a conversion checking algorithm for dependent types. In general, type equality is determined through their conversion into a normal form and subsequent verification of α or syntactic equality [14].

As mentioned by Gratzer, Sterling, and Birkedal, the naïve approach for implementing a conversion checker then would be reducing each term until it is no longer possible, and then comparing them for syntactic equality [5]. Due to efficiency concerns, more sophisticated techniques that do not use reduction tactics alone are utilised.

This section provides an overview of the techniques found in this exploratory literature survey. A brief overview of how the implementation works is provided, along with some cited advantages and disadvantages of the system. This section covers the different techniques in the following order...

5.1 Naïve Substitution

The simplest technique for conversion checking is naïve, capture-avoiding substitution, as used in Lean's kernel [15; 16]. This approach aims to obtain normal forms through term substitution alone.

Despite its simplicity in terms of theory and implementation, it has many drawbacks. Ultimately, it is a very slow approach, requiring other techniques such as memoization to maintain good performance, ditracting from the initial simplicity of the algorithm by adding complexity [16].

5.2 Normalisation by Evaluation

Normalisation by Evaluation (NbE) is a technique that can be used to determine definitional equality in dependent type theories. It works by first transforming a term in a semantic model of a language, and then reifying it back into a term representation, thus obtaining a normal form [17, p. 25].

For an in-depth look into NbE, its different variants and history, Abel's habilitation thesis [18] is recommended. Although it is from 2013, it provides some valuable insight into the subject.

Despite favourable theoretical properties and improved efficiency over a naïve substitution approach, NbE is not always adequate for certain languages. For example, the fact that it η-expands terms until it is no longer possible, means that it is sometimes not what is necessary for a language. Occasionally, it is not even possible within a certain type theory without alterations [19; 17, p. 16]

Concerns about NbE's efficiency have also been brought up. Fully normalising expressions is not always necessary for determining type equality, and sometimes a simple α-equivelence check is sufficient [19]. However, this can often be mitigated by combining this approach with a simple algorithm that checks if two terms are syntactically identical before performing NbE.

Normalisation by Evaluation also comes in two flavours - typed and untyped NbE. While typed NbE generally has higher performance, untyped NbE can favoured due to its generalisability [20]. However, this comes with significant overhead due to the necessity of adding tags to object syntax interpretation to embed the approach into a new language. This can be mitigated to an extent, but not completely [21].

5.3 Using Abstract Machines

Using abstract machines has been proposed as a method of making a conversion checker more efficient [22]. This approach is used by Agda[3] [23] and Coq[4] [24], both languages that implement dependent types. This subsection focuses on these two approaches, but other similar implementations exist, such as the technique proposed by Kleeblatt [17].

Agda Abstract Machine

The implementation of Agda's typechecker uses an abstract machine for compile-time reduction, as can be seen in the source code [23].

This implementation uses a call-by-need machine, and was found to offer better efficiency than their slow, substitution-based, strategy. However, the comments in the code state that the type checker would fall back to slow reduction occasionally, especially when a definition is encountered that is not supported by the machine. Nevertheless, a recent fix has been implemente to ensure that the fast reduction strategy is always used.

Unfortunately, as there is no published literature on this technique, it is difficult to analyse the implementation technique based solely on the code and accompanying comments. However, it is still interesting to note that this technique exists, and does seem to offer some advantages over not using an abstract machine for conversion checking.

Strong Reduction with the Zinc Abstract Machine

An alternative implementation of conversion checking with an abstract machine was proposed by Grégoire and Leroy in 2002 [22]. This technique for strong reduction is one that is currently used in Coq's `vm_compute` tactic [24].

Although this is not mentioned in the paper itself, it has been argued by other researchers that this is an instance of untyped NbE [18, p. 67; 20].

This method utilises a modified version of the Zinc Abstract Machine (ZAM) that was proposed by Leroy [25]. This is an abstract machine that uses a call-by-value evaluation strategy, unlike the machine discussed for the Agda Abstract Machine. For conversion checking, terms are compiled down to the bytecode of the ZAM, which performs the strong normalisation through compiled weak symbolic reduction and an interpreted recursive readback procedure [22].

This strategy is reported to have a few benefits. Coq's documentation states that it is especially useful for the full evaluation of algebraic objects [24]. However, the main advantage of this strategy lies in its efficiency.

Implementing this technique in Coq has lead to significant improvements in Coq's efficiency. It was on average signifi-

[3]https://agda.readthedocs.io

[4]https://coq.inria.fr/

cantly faster than Coq's `cbv` and `lazy` strategies[5], as can be seen in the benchmarks provided by Grégoire and Leroy [22]. Kleeblatt suggested that this was because the strategy used strict evaluation, which is often efficient, and issues of non-termination were avoided by the finite nature of reduction sequences of Coq's well-typed terms [17]. Thus, the efficiency of this algorithm can be said to be due to both its intrinsic efficiency, and its pairing with a favourable language.

This implementation strategy is not without drawbacks. According to Coq's documentation, it cannot be fine-tuned [24], which can limit the type checker and/or language. Additionally, while it is often an efficient strategy, Grégoire and Leroy's benchmarks demonstrated a small slow-down over the `cbv` and `lazy` strategies [22]. This was explained by the fact that the compilation to bytecode happens every time with this technique. This is disadvantegeous when the terms to be compared are already in normal form, as nothing is gained but the price of compiling to bytecode must still be paid. While it appears from the benchmarks that the overhead is small enough to be acceptable, it is still something that implementers may want to keep in mind.

5.4 Normalisation by Translation to OCaml

Another approach to implementing strong normalisation similar to that in Grégoire and Leroy [22] was proposed by Boespflug, Dénès, and Grégoire in 2011 [20]. It is currently used as one of Coq's performance-oriented reduction strategies `native_compute`, alongside `vm_compute` [24].

The implementation details are similar to the Strong Reduction with the Zinc Abstract Machine (SRZAM) technique described in subsection 5.3. This is in large part due to it being created by building upon the SRZAM approach [20]. However, the main difference is that instead of compiling to an abstract machine, it translates terms directly into OCaml.

This approach has a few advantages over using the Zinc Abstract Machine in Grégoire and Leroy's approach. One such advantage is in efficiency. According to Coq's documentation, it is 2-5 times faster than the SRZAM approach [14]. The explanation offered by the authors Boespflug, Dénès, and Grégoire is that by compiling to native code, the approach avoids the limitations on efficiency placed by the abstract machine [20]. Additionally, the direct conversion to OCaml means that this approach does not need to work with a custom version of the ZAM. This is beneficial because an abstract machine does not need to be maintained separately, and it creates a better separation of concerns. Overall, it is evident that a direct conversion to OCaml has some benefits over the SRZAM approach.

In addition to being performant, this approach also aims to be portable and easily generalisable to other (functional) languages. As an instance of untyped NbE [18, p. 67], it attempts to include the generality of untyped NbE while matching the performance of typed NbE [20].

However, there are also disadvantages to compiling directly to OCaml. As this approach has significantly more overhead than the SRZAM appraoch, Coq often recommends that SRZAM still be used over this technique [14].

[5]https://coq.inria.fr/refman/proofs/writing-proofs/equality.html#applyingconversionrules

5.5 Equality through "Shape" Analysis

As an alternative to reduction or normalisation-based equality, it is also possible to determine type equality through the "shapes" of terms. This is often done because always reducing terms to normal form before checking for equality, as done in NbE, is often wasteful. This section will look at two implementations using this type of technique - the original algorithm, as well as a unique spin on it that includes dependency erasure. However, there are other implementations that are worth investigating, such as another refinement by Abel and Scherer [26] and a popular formalisation by Abel, Öhman, and Vezzosi in 2018 [6].

Original Algorithm by Coquand, 1991

While there are generally no significant problems in implementing an algorithm for β-reduction for languages implementing dependent types, doing so for $\beta\eta$-reduction is often much more complex [28]. The method proposed by Coquand in 1991 avoids this completely by deciding equality directly through analysing the "shapes" of terms, using the principle of extensionality instead of η reduction or expansion [27; 28].

The main benefit of this technique is that it avoids explicit normal form computation, which has been an issue with previous techniques. This means that the algorithm will terminate early if terms are found to be unequal [28].

However, this is an old proposed technique, and thus has some significant disadvantages. One of these is that it is not easily scalable to richer type theories due to its strong reliance on the "shape" of terms [28]. Therefore, considerations should be made to newer refinements of this technique that preserve the benefits while making it as scalable as possible.

Equality by Dependency Erasure and "Shape" Analysis

An interesting development on Coquand's original algorithm was the type-directed approach created by Harper and Pfenning for a subset of the Edinburgh Logical Framework (LF). Unlike what was originally proposed by Harper, Honsell, and Plotkin [29], and the Coquand's algorithm, this technique considers both η and β conversion.

This technique uses multiple ideas for conversion checking, one which is the erasure of dependencies when deciding equality, despite the presence of dependently-typed terms [28]. This is perhaps the first equality algorithm that implements such a technique. This type-directed equality algorithm therefore only requires testing equality on simple types.

However, as this technique was inspired by Coquand, it also includes a focus on the "shape" of the terms [28]. This is used in the first phase of the implementation to determine the approximate non-dependent types, to provide information for the verification that dependencies are respected throughout the technique.

The authors of this technique consider it to be an efficient implementation. This is because the usage of approximate types leads to fast equality and unification operations, as the computation and type-checking of precise types is often expensive Harper and Pfenning. However, no benchmark results are provided, and this judgement is purely subjective.

Another advantage that this technique provides is scalability to richer languages. While this was developed for a subset of LF, a blueprint for adapting the method to other type theories is provided by Harper and Pfenning [28].

Nevertheless, it cannot scale to all languages. The authors have stated that it will handle poorly in an impredicative setting or with predicative universes [28]. While a way for handling the case of predicative universes has been suggested by the authors, implementing it for impredicative settings is likely more difficult. Additionally, it is sometimes impossible to eliminate dependencies completely. This is the case with singleton kinds and subkinding, although some alterations such as those proposed by Stone, College, and Harper can be made to create a similar technique for them [30]. Some authors also report that this technique does not permit the definition of types by recursion on values [26], which is a feature that is common within many proof assistants. As can be seen, there are many situations in which a pure version of this technique cannot be used for conversion checking within a language.

5.6 Normalisation by Hereditary Substitution

Another technique is Normalisation by Hereditary Substitution (NbHS), which was first introduced by Watkins et al. [31]. This technique includes the simultaneous performance of syntactical substitution and normalisation of terms [32]. Abel described hereditary substitutions as the *"substitution of a normal form into another one, triggering new substitutions to remove freshly created redexes, until a normal form is returned."* [33].

This technique also has some advantages, such as its structurally recursive nature [33]. Additionally, hereditary substitutions preserve canonical forms during substitution [32], which can be valuable for some implementations.

However, this exploratory survey has not found this to be a popular technique for conversion checking. In discussion forums, it has been noted that there is no reason to use hereditary substitution unless there is a particular interest in reverse mathematics, as most other techniques are more favoured, and hereditary substitutions can be formally difficult to handle [34].

5.7 Conversion Checking in the Presence of Non-Termination

Non-termination is a feature that languages implementing dependent types may want to implement. However, this causes significant complications to the conversion checking algorithm in the presence of dependent types. This section will look at two approaches to dealing with the conversion checking of dependent types in the presence of non-termination - that of Dependent Haskell, and ZOMBIE.

Dependent Haskell

Dependent Haskell takes an unusual approach to handling non-termination through keeping the type-checking undecidable [35, p. 66]. Therefore, more responsibility is placed fully on the programmer - if the type checker does not terminate (or alternatively, is taking an unusually long time), they will have to terminate it and make their own assumptions on what went "wrong".

Non-Termination with ZOMBIE

A language that takes a different approach to type checking than Dependent Haskell is ZOMBIE, which was developed by Sjöberg [36, p. 6]. ZOMBIE is a language that was developed as a part of the Trellys Project [4, p. 6], whose purpose was to create a functional programming language that included general recursion as well as full dependent types [37].

Unlike Dependent Haskell, the functional language ZOMBIE retains decidable type-checking by excluding automatic β-conversion completely [38]. While checking for β-equivalence is still possible, this technique forces the user to indicate how much to β-reduce their types.

As an alternative to the standard equality that is used, ZOMBIE utilises a "congruence closure" relation to define type equality. This was done because efficient algorithms already exist to decide congruence closure, making decidability of type checking much simpler to ensure.

6 Discussion and Advice

Overall, this exploratory survey has demonstrated that there are a lot of different techniques to choose from, and there is not necessarily a "best" technique that fits all languages or ideas. However, this section will aim to provide implementers with some advice on which technique may suit their idea of a conversion checker better.

Portability One important facet to consider in a conversion checker is its portability. While some implementers may be willing to make more modifications to a conversion checking algorithm to embed it in their language, this is not always the case. However, that as can be seen with untyped NbE, generalisability and portability can often come with the drawback of low performance. However, there are still techniques that manage to preserve both performance and portability, such as the SRZAM technique discussed in 5.3, and the technique detailed in section 5.4. However, in general, more performant techniques tend to require more modifications for proper integration.

Extendibility If a less portable solution is chosen, the extendibility of it to richer type systems must be considered. Many techniques discussed in this survey have been developed for specific (and occasionally subsets of) type theories. This can mean that some techniques that have been discussed, such as those mentioned in subsection 5.5 can not scale easily or at all to all type theories, or support certain desirable features.

Simplicity The ease of implementing the conversion checker may also be taken into account by implementers, especially when performing the implementation from scratch. However, simple solutions such as Naïve Substitution can often come with drawbacks such as a lack of efficiency, and the resulting complexity from optimising the type checker to maintain performance.

Efficiency The efficiency of the conversion checking algorithm is also important. This is because programmers may dislike languages with noticeable slow compile times due to a slow conversion checker, and this can reduce the acceptance and usage of the language. Oftentimes, to find more efficient implementations, later derivative works need to be looked at. However, this efficiency can often come with drawbacks - for example, although the technique described in subsection 5.4 has been noted to be significantly more efficient, the SRZAM technique that it is based on (discussed in 5.3) is still used and often recommended in Coq. Nevertheless, given the current state of literature it is difficult to provide objective comparisons on the efficiency of different techniques - unlike the study done by Boespflug, Dénès, and Grégoire [20], most studies do not provide comparative benchmarks. Therefore, this exploratory survey cannot provide definitive advice on the efficiency of most techniques in comparison to each other.

In the event that a less efficient technique is chosen, implementers should note that a lot of techniques can be used in conjunction, if they are willing to make modifications. Techniques such as memoisation, or even performing a simple α-equivalence check before doing more expensive computations can offer significant speed-ups to the algorithm. Thus, it may be not as important to focus on the efficiency of an individual technique, but also look at additional optimisations that can be done to it.

Abstract Machines The usage of an abstract machine should also be considered. From subsection 5.3, it is evident that the use of abstract machines for conversion checking is being used successfully in at least two popular languages that use dependent types. Unfortunately, while these two techniques that use abstract machines report improvements in efficiency, this survey could not find discussions on the use of an abstract machine for conversion checking without an associated technique. Therefore, it is difficult to identify whether using an abstract machine is beneficial over implementing a similar algorithm without an abstract machine. It has also been argued that the use of an abstract machine can limit the efficiency of an implementation, as it is dependent on the abstract machine which is often less efficient than native code. In such cases, compilation to native code as described in 5.4 has been cited to be a more efficient solution. Additionally, there are drawbacks that may reduce the implementability of this technique for a new or existing language. If a modified version on an abstract machine is used, it creates the additional concern of keeping it up-to-date. However, modifications might sometimes be necessary, as some definitions that are wanted in a language may not be supported by the default abstract machine. As can be seen with Agda, this can be worked around by falling back on a slower conversion-checker, but that would result in two conversion checkers having to be maintained, which may be undesirable. Therefore, although the examples discussed demonstrate that using an abstract machine may be efficient, this should not be taken at face value.

Language Features Finally, the implementer must consider what features they want from the language they are implementing, as some features can have a strong impact on the chosen conversion checking algorithm. For example, the presence of non-termination, as discussed in subsection 5.7, can have drastic consequences on the conversion checker. This is because the implementation will have to either leave type checking undecidable, or work with unorthodox workarounds as in ZOMBIE.

However, this decision will also depend on whether or not the implementer values the decidability of their type checker. While enforcing decidability of conversion checking can compromise the expressivity of a type theory or language, the decidability ensures ease of use and minimisation of developer confusion through guaranteed termination [39]. As put by Nawrocki, an undecidable type checker can create the question of *"When our dependently-typed program is taking a long time to check, is the compiler just slow or is it trying to prove the Riemann hypothesis?"*. However, it has also been argued by authors such as Eisenberg that programs looping at run-time is not unusual, and programmers are not surprised by this occurrence. By this logic, a program looping at compile-time due to undecidable type-checking should not be too shocking [35]. Additionally, suggestions have been made to halt the type-checking with an error message if the number of reduction steps gets too high to give a better user experience [40; 35, p. 66]. Hence, while decidability of the conversion checking algorithm is often a favourable quality, perhaps it is not necessary for it to perform to a good standard for the programmer.

6.1 Limitations

Before making conclusions, it is important to discuss some of the limitations of this exploratory survey. This is to ensure that the findings are not misrepresented, and show points of improvement for future research.

Given the nature of this paper being an exploratory survey of different implementation techniques, as well as the current state of the literature, most judgements and advice given in this paper are subjective. For example, although many techniques state that their technique is "efficient", it is difficult to objectively compere this efficiency between techniques. Benchmarks are rare, with some limited exceptions, in this field. Unfortunately, there is a chance that some techniques that have unjustifiably self-reported themselves as "efficient" have been labelled as such in this paper.

There is also a chance that not all techniques have been identified. This is both due to the limited time-frame of this paper, and the current literature often utilising inconsistent terminology.

Additionally, not all sub-variants of different techniques have been covered in this paper. This is because there are a vast amount of such minute variations on covered techniques, and it is impossible to cover them all in this short exploratory survey. While the most stand-out modifications have been covered, there is a chance that a specific modification that an implementer is looking for exists in literature, but has not been discussed or identified in this survey.

In conclusion, this paper possesses many limitations that should be kept in mind when considering the advice and conclusions provided.

7 Conclusions and Future Work

In conclusion, this exploratory survey identified several different techniques that can be used to implement a type checker. For normalisation into normal forms, Naïve Substitution, Normalisation by Evaluation (NbE), Equality through "Shape" Analysis, and Normalisation by Hereditary Substitution (NbHS) were the main tactics. Naïve Substitution is the simplest implementation, as it uses only term substitutions in the base algorithm. While NbE avoids direct reduction through transformations and reificatoins, Equality through "Shape" Analysis looks at the general "shapes" of terms. NbHS, however, simultaneously normalises and performs syntactical substitution on terms to achieve a normal form. Equality through congruence closure was also looked at in 5.7, which avoids β-conversion entirely, and maintains decidable conversion checking in the face of non-termination. It was also found that, although most languages prefer to incorporate decidable type checking, there are languages such as Dependent Haskell which, due to the difficulties of conversion checking under non-termination, choose to have type checking be undecidable. Implementations that focused more on efficiency were also identified, which either utilised an abstract machine, or compiled terms into another language such as OCaml.

Each of these tactics has advantages and disadvantages. Differing advice is given to implementers depending on the portability, extendability, simplicity, efficiency, and decidability that they want from the conversion checker, as well as the desired functionality of the language. However, it should be noted the judgements on features such as extendablity and portability are subjective.

7.1 Future Work

As identified in this exploratory survey, there are many gaps in the literature that can be filled with future work.

One such gap is that this paper does not pay particular attention to what type of equality is used. It may be important for developers to consider which and what amount of different equalities to implement, as more equalities can make the type theory more powerful [41, p. 54]. Thus, further investigations can be made that analyse the effects and benefits of including or excluding equalities.

Additionally, as discussed in Limitations subsection, for efficiency of the implementation techniques, this paper can give limited advice as the techniques have not been fully compared yet. While some papers offer comparative benchmarks, this is not standard. A comprehensive overview of the efficiency of certain techniques, with all of them being tested to measure performance, can thus be done in the future.

Some implementation techniques also do not have a paper describing the implementation and the theory behind it. For example, this is evident with Agda's conversion checker using the Agda Abstract Machine, or the conversion checker in Lean's kernel. This makes reasoning about, and implementing such techniques difficult. Filling such gaps in research could be a topic for future research.

Finally, this paper had a strong focus on conversion checking algorithms for dependent types. There may be some overlap with other types or features that implementers would want to have in their language, but would significantly affect the conversion checking algorithm. Looking into this to make sure that the advice remains the same, or alternatively showing why the advice no longer holds, could be valuable.

Overall, there are many avenues within which more research can be done to provide a better overview of the different conversion checking algorithms for dependent type theories.

References

[1] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1.

[2] Ana Bove and Peter Dybjer. "Dependent types at work". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5520 LNCS (2009), pp. 57–99. ISSN: 03029743. DOI: 10.1007/978-3-642-03153-3_2.

[3] Samuel Baxter. "An ML Implementation of the Dependently Typed Lambda Calculus". Honors Thesis. Boston College, May 2014.

[4] Vilhelm Sjöberg. "A Dependently Typed Language with Nontermination". In: *Publicly Accessible Penn Dissertations* (2015). URL: https://repository.upenn.edu/edissertations/1137.

[5] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. "Implementing a modal dependent type theory". In: *Proceedings of the ACM on Programming Languages* 3 (ICFP Aug. 2019). ISSN: 24751421. DOI: 10.1145/3341711. URL: https://doi.org/10.1145/3341711.

[6] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. "Decidability of conversion for type theory in type theory". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018). ISSN: 24751421. DOI: 10.1145/3158111.

[7] nLab authors. *alpha-equivalence*. Revision 7. June 2023. URL: https://ncatlab.org/nlab/show/alpha-equivalence (visited on 06/15/2023).

[8] Roy L. Crole. "Alpha equivalence equalities". In: *Theoretical Computer Science* 433 (2012), pp. 1–19. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2012.01.030.

[9] Benno van den Berg and Martijn den Besten. "Quadratic type checking for objective type theory". In: *CoRR* abs/2102.00905 (2021). arXiv: 2102.00905. URL: https://arxiv.org/abs/2102.00905.

[10] nLab authors. *beta-reduction*. Revision 6. June 2023. URL: https://ncatlab.org/nlab/show/beta-reduction (visited on 06/15/2023).

[11] Milo Davis. *Inside PRL - Beta Reduction (Part 1)*. Nov. 2016. URL: https://prl.khoury.northeastern.edu/blog/2016/11/02/beta-reduction-part-1/ (visited on 06/15/2023).

[12] Luke Palmer. *[Haskell-cafe] What's the motivation for η rules?* Dec. 2010. URL: https://mail.haskell.org/pipermail/haskell-cafe/2010-December/087783.html (visited on 06/15/2023).

[13] nLab authors. *eta-conversion*. Revision 13. June 2023. URL: https://ncatlab.org/nlab/show/eta-conversion (visited on 06/15/2023).

[14] *Conversion rules — Coq 8.17.0 documentation*. URL: https://coq.inria.fr/refman/language/core/conversion.html (visited on 06/12/2023).

[15] Leonardo de Moura et al. *lean4/src/kernel/type_checker.cpp at master · leanprover/lean4*. Jan. 2023. URL: https://github.com/leanprover/lean4/blob/master/src/kernel/type%5C_checker.cpp (visited on 06/24/2023).

[16] András Kovács, Sebastian Ullrich, and Vladislav Zavialov. *AndrasKovacs/smalltt: Demo for high-performance type theory elaboration*. Apr. 2023. URL: https://github.com/AndrasKovacs/smalltt/tree/master (visited on 06/24/2023).

[17] Dirk Kleeblatt. "On a Strongly Normalizing STG Machine with an Application to Dependent Type Checking". PhD Thesis. Technical University of Berlin, Apr. 2011. DOI: 10.14279/depositonce-2798. URL: https://doi.org/10.14279/depositonce-2798.

[18] Andreas Abel. "Normalization by Evaluation: Dependent Types and Impredicativity". Habilitation Thesis. Ludwig-Maximilians-Universität München, 2013. URL: http://www.cse.chalmers.se/~abela/habil.pdf.

[19] David Thrane Christiansen. *Checking Dependent Types with Normalization by Evaluation: A Tutorial*. URL: https://davidchristiansen.dk/tutorials/nbe/ (visited on 06/12/2023).

[20] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full reduction at full throttle". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7086 LNCS (2011), pp. 362–377. ISSN: 03029743. DOI: 10.1007/978-3-642-25379-9_26.

[21] Mathieu Boespflug. "Conversion by Evaluation". In: *Practical Aspects of Declarative Languages*. Ed. by Manuel Carro and Ricardo Peña. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 58–72. ISBN: 978-3-642-11503-5. DOI: 10.1007/978-3-642-11503-5_7.

[22] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: ACM, Sept. 2002, pp. 235–246. ISBN: 1581134878. DOI: 10.1145/581478.581501.

[23] Ulf Norell et al. *agda/Fast.hs at master · agda/agda*. May 2023. URL: https://github.com/agda/agda/blob/master/src/full/Agda/TypeChecking/Reduce/Fast.hs (visited on 06/12/2023).

[24] *Reasoning with equalities — Coq 8.17.0 documentation*. URL: https://coq.inria.fr/refman/proofs/writing-proofs/equality.html (visited on 06/12/2023).

[25] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. INRIA, Feb. 1990. URL: https://inria.hal.science/inria-00070049.

[26] Andreas Abel and Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". In: *Logical Methods in Computer Science* 8 (1 Mar. 2012), pp. 1–36. ISSN: 1860-5974. DOI: 10.2168/LMCS-8(1:29)2012. URL: https://lmcs.episciences.org/1045.

[27] Thierry Coquand. "An algorithm for testing conversion in type theory". In: *Logical Frameworks*. Book has been checked out. 1991, pp. 255–279. ISBN: 0 521 41300 1.

[28] Robert Harper and Frank Pfenning. "On equivalence and canonical forms in the LF type theory". In: *ACM Transactions on Computational Logic (TOCL)* 6 (1 Jan. 2005), pp. 61–101. ISSN: 15293785. DOI: 10.1145/1042038.1042041.

[29] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the ACM* 40 (1 Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060. URL: https://dl.acm.org/doi/10.1145/138027.138060.

[30] Christopher A Stone, Harvey Mudd College, and Robert Harper. "Extensional Equivalence and Singleton Types". In: *ACM Transactions on Computational Logic* 7 (4 2006), pp. 676–722.

[31] Kevin Watkins et al. "A concurrent logical framework: The propositional fragment". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3085 (2004), pp. 355–377. ISSN: 16113349. DOI: 10.1007/978-3-540-24849-1_23.

[32] Chantal Keller and Thorsten Altenkirch. "Hereditary Substitutions for Simple Types, Formalized". In: Sept. 2010. URL: https://inria.hal.science/inria-00520606.

[33] Andreas Abel. "Implementing a normalizer using sized heterogeneous types". In: *Journal of Functional Programming* 19 (3-4 July 2009), pp. 287–310. ISSN: 1469-7653. DOI: 10.1017/S0956796809007266.

[34] András Kovács. *What is hereditary substitution and why should I use it?* Mar. 2018. URL: https://proofassistants.stackexchange.com/questions/1174/what-is-hereditary-substitution-and-why-should-i-use-it/1177#1177 (visited on 06/12/2023).

[35] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". PhD Thesis. University of Pennsylvania, Oct. 2016. URL: http://arxiv.org/abs/1610.07978.

[36] Chris Casinghino. "Combining proofs and programs". In: *Dissertations available from ProQuest* (2014). URL: https://repository.upenn.edu/dissertations/AAI3670881.

[37] Stephanie Weirich and Matúš Tejiščák. *sweirich/trellys*. July 2019. URL: code.google.com/p/trellys (visited on 06/15/2023).

[38] Vilhelm Sjöberg and Stephanie Weirich. “Programming up to Congruence”. In: ACM, Jan. 2015, pp. 369–382. ISBN: 9781450333009. DOI: 10.1145/2676726.2676974. URL: https://dl.acm.org/doi/10.1145/2676726.2676974.

[39] Wojciech J Nawrocki. “Decidability of typechecking in a dependently-typed programming language with sigma types”. Master Thesis. University of Cambridge, June 2020.

[40] Lennart Augustsson. “Cayenne — A Language with Dependent Types”. In: *Advanced Functional Programming*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 240–267. ISBN: 978-3-540-48506-3.

[41] David Aspinall and Martin Hofmann. “Dependent Types”. In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. The MIT Press, Dec. 2004, pp. 45–86. ISBN: 9780262281591. DOI: 10.7551/mitpress/1104.003.0004.