Towards a Syntactic Model of Sized Dependent Types

JONATHAN CHAN, University of British Columbia, Canada, jcxz@cs.ubc.ca
Graduate student (MSc.) advised by William J. Bowman

1 TERMINATION CHECKING FOR DEPENDENT TYPE THEORIES

The types-as-propositions paradigm associates certain type theories with formal logical systems, and consequently types in those theories with propositions in those logics. Furthermore, well-typed programs are associated with proofs of the corresponding proposition. Many dependent type theories, for instance, correspond to higher-order logics, and having an automated type checker means having the ability to automatically verify proofs.

One must be careful, however, not to allow nonterminating programs, because they correspond to logical inconsistencies, i.e. proofs of falsehood. Additionally, in dependent type checkers where programs may be evaluated during type checking, failure to rule out nonterminating programs leads to nonterminating type checking. Contemporary proof assistants based on dependent type theories, such as Coq, Agda, Lean, Idris, and many more, typically restrict recursive functions to *structurally-recursive* ones, where the argument of recursive calls must be *syntactically* smaller, peeling away layers of constructors until a base case is reached. Type checkers in these proof assistants use *guard predicates* [8] to ensure the restriction.

However, the guard predicate is often *too* restrictive to accept a variety of recursive functions for which termination is otherwise evident to the discerning programmer. In particular, functions recurring on subarguments that have first been passed to other functions known not to add any more layers of constructors must surely terminate, but since the recursive argument is not *syntactically* the subargument, the guard predicate does not hold.

Some type checkers will inline function definitions for the purpose of termination checking, but this reliance on other function definitions makes code non-modular, and inlining very large definitions could severely negatively impact type checking performance. Furthermore, the syntactic nature of the guard predicate makes it sensitive to minor syntactic changes, and a subtle refactoring of a function inlined in later functions could affect whether those functions even pass termination checking at all! In short, a syntactic guard predicate goes against good programming practices.

2 TYPE-BASED TERMINATION CHECKING

An alternative to syntactic termination checking is to instead use *type-based* termination checking, where if a recursive function type checks without involving any other termination conditions, then it is guaranteed to terminate. One such method uses *sized types* [11], where inductive types carry additional size information. Intuitively, the size is a measure of how many layers a member of that type contains, and constructors must have a greater size than its subarguments. The types of functions then carry information about whether it affects the size of its argument, meaning that no inlining is required — only the type is needed, not the whole definition.

Sized types have been implemented in Agda and can be enabled with the `--sized-types` pragma.[1] It encludes sophisticated features like first-class sizes and bounded size quantification. There is also a large body of theoretical work on sized types in various type systems, but none of them quite satisfy all of the desirable features.

[1]Unfortunately, the implementation is inconsistent due to the presence of an *infinite size*, which is defined to be the size strictly greater than all other sizes, including itself.

- Barthe et al. [5], Grégoire and Sacchini [9], Sacchini [15], and Sacchini [16] introduce and prove consistent a lineage of Calculi of (Co)Inductive Constructions (CIC) with sized types, but only prenex size quantification is possible: one cannot, for instance, pass around a higher-order function quantifying over a size.
- Abel [1], Abel [2], and Abel and Pientka [3] introduce not only higher-rank size quantification but also bounded size quantification, the latter of which eliminates the need for complex monotonicity checks or syntactic approximations thereof. However, these type systems extend System F_ω rather than a dependent type theory.
- Abel et al. [4] prove normalization of a higher-rank sized dependent type theory with naturals, but without bounded size quantification.

In ongoing work, I seek to prove the logical consistency of Sized CC_ω, a higher-rank sized dependent type theory with bounded size quantification. Rather than using very involved set-theoretic methods like in Sacchini's dissertation [15] or the normalization by evaluation technique in Abel et al. [4] which requires a typed definitional equality judgement in the type theory, I instead define a *syntactic model* [6] into Extensional CIC (CIC_E) [14]. That is, I need to define a compiler from Sized CC_ω to CIC_E, then prove that it is *type-preserving*: given some well-typed term in Sized CC_ω, if both the term and its type are translated to CIC_E, then the translated term should also be well typed against the translated type. Because CIC_E is known to be consistent, and an inconsistency in Sized CC_ω implies the existence of an inconsistency in CIC_E via the type-preserving compilation, inconsistency of Sized CC_ω would be a contradiction.

3 SYNTACTIC MODEL OF SIZED CC_ω

Sized CC_ω is a Generalized Calclulus of Constructions with definitions ($CC\omega$) [10] — that is, a Calculus of Constructions with untyped equality, a cumulative universe hierarchy, and `let` expressions — extended with bounded and unbounded size quantification, abstraction, and application, as well as size expressions consisting of size variables, a base size, and a size successor operation. I further add naturals and W types only, but these should scale directly to inductive types in general.

In Sized CC_ω, the natural type and W types are parametrized by some size, and their constructors quantify over a bounded size representing the strictly smaller size of recursive subarguments. In CIC_E, I define a `Size` inductive type representing the sizes in Sized CC_ω, and an indexed inductive type `_⩽_` on `Size` representing the ordering relation used in bounded quantification and abstraction. The natural type and W types then compile to corresponding inductive types literally parametrized by `Size`, and whose constructors take proofs of strict inequality of two `Size`s.

The majority of the remaining translation is straightforward, especially for universes, functions, `let` expressions, and `case` expressions. Bounded size quantification and abstraction correspond to quantification and abstraction over a `Size` and an inequality, and correspondingly for unbounded ones. But what about fixpoints?

The typing rule for fixpoints in Sized CC_ω has as premise the well-typedness of its body in an environment where the fixpoint itself is in scope, but quantifying over a smaller size. The key insight is that fixpoints now correspond to *well-founded induction* over sizes, rather than structural induction. To show that well-founded induction indeed holds for `Size`, I first show that all `Size`s satisfy an *accessibility predicate* [13]; well-founded induction then follows by a structurally-inductive proof over the predicate. Fixpoints in Sized CC_ω then translate immediately to applications of well-founded induction.

Now that a translation from Sized CC_ω to CIC_E is established, I show that it is type preserving. Because Sized CC_ω uses an untyped equality judgement, I can use standard techniques for showing type preservation [7]. An important proof detail is that equality reflection (and therefore extensionality) is required to show an η-equivalence rule for `case` expressions and to show that proofs of accessibility are equal, which are properties used to prove that the translations of an applied fixpoint and its reduction in Sized CC_ω are definitionally equal in CIC_E.

4 STATUS AND FUTURE WORK

The work is not yet done; there remain unresolved problems with the model, and additional features to add that one would expect from a practically-useable sized dependent type theory.

4.1 Universe Levels and Size

To be able to assign sizes to *general inductive types* such as W types, which conceptually can have transfinitely many recursive subarguments, `Size` itself must be able to express the same transfinitivity. Therefore, its inductive definition in CIC_E mirrors that of *Brouwer ordinals* [12], although the domain of the function in the size corresponding to the limit ordinal is an arbitrary type A rather than merely the usual natural numbers. `Size` itself must then live in a universe higher than that of A, according to the usual well-formedness restrictions on inductive types.

Recall that the natural type and W types in Sized CC_ω are parametrized by `Size`. Given a W type with type parameters $A : \mathtt{Type}_\ell$ and $B : A \to \mathtt{Type}_\ell$, the type used in limit sizes for the W type would also be A. Meanwhile, the naturals aren't transfinite, so we simply have $A := \bot : \mathtt{Type}_0$, the uninhabited type. Unfortunately, since `Size` itself would then live in $\mathtt{Type}_{\ell+1}$ and $\mathtt{Type}_1$, respectively, so must the W type and the natural type, rather than in $\mathtt{Type}_\ell$ and $\mathtt{Type}_0$ as one would expect. Intuitively, `Size` itself must be "large enough" (in the type universe sense) to include all sizes of naturals and elements of W types, which makes it "too large" to live in the same universe as what it should include.

One unsatisfactory solution would be to accept the natural type and W types living in larger universes than they normally would in an unsized dependent type theory. Another solution would be to parametrize `Size` itself by the limit size's type A, which would allow it to live in the same universe as A. However, the translation of sizes and size quantifications and abstractions would have an underdetermined parameter, and sizes used for one inductive could not be used for another.

4.2 The Infinite Size

In nearly all past work on sized types, including the Agda implementation, there is a notion of an infinite size ∞ that is strictly larger than all sizes, including itself: the relation $\infty < \infty$ holds. Sized CC_ω does not have the infinite size, because this property would make sizes no longer well-founded, undermining all efforts to interpret fixpoints as applications of well-founded induction. In fact, this is why sized types are inconsistent in Agda: dependent types make it possible to internalize the order on sizes as an inductive type within Agda itself, from which well-foundedness can be proven, yielding falsehood when combined with $\infty < \infty$. Finding a suitable replacement for uses of ∞ that capture its convenience while retaining consistency remains an open problem. One possibility is to use an existentially size-quantified inductive type in place of the ∞-sized inductivebut it appears this might require a nonconstructive axiom that does not compute.

4.3 Coinductive Types

Aside from termination checking, sized types are also used for *productivity checking* of *corecursive* definitions, making reasoning about corecursive constructions much easier. If Sized CC_ω is indeed consistent, I expect that extending the language and the proofs to include sized coinductive types would be relatively straightforward.

REFERENCES

[1] Andreas Abel. 2006. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types.* Theses. University of Munich. http://www.cse.chalmers.se/~abela/diss.pdf

[2] Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. *Electronic Proceedings in Theoretical Computer Science* 77 (Feb 2012), 1–11. https://doi.org/10.4204/eptcs.77.1

[3] Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* 26 (2016), e2. https://doi.org/10.1017/S0956796816000022

[4] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110277

[5] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2006. $\widehat{\text{CIC}}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, Proceedings (Lecture Notes in Artificial Intelligence, Vol. 4246)*, Hermann, M and Voronkov, A (Ed.). Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany, 257–271. https://doi.org/10.1007/11916277_18

[6] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, Paris, France, 182–194. https://doi.org/10.1145/3018610.3018620

[7] William J. Bowman. 2018. *Compiling with Dependent Types.* Theses. Northeastern University. https://doi.org/10.17760/D20316239

[8] Eduardo Giménez. 1995. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, Peter Dybjer, Bengt Nordström, and Jan Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–59.

[9] Benjamin Grégoire and Jorge Luis Sacchini. 2010. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Christian G. Fermüller and Andrei Voronkov (Eds.). Vol. 6397. Springer Berlin Heidelberg, Berlin, Heidelberg, 333–347. https://doi.org/10.1007/978-3-642-16242-8_24 Series Title: Lecture Notes in Computer Science.

[10] Robert Harper and Robert Pollack. 1991. Type checking with universes. *Theoretical Computer Science* 89, 1 (Oct. 1991), 107–136. https://doi.org/10.1016/0304-3975(90)90108-T

[11] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

[12] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. 2021. Connecting Constructive Notions of Ordinals in Homotopy Type Theory. In *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 202)*, Filippo Bonchi and Simon J. Puglisi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 70:1–70:16. https://doi.org/10.4230/LIPIcs.MFCS.2021.70

[13] Bengt Nordström. 1988. Terminating general recursion. *BIT Numerical Mathematics* 28, 3 (1988), 605–619. https://doi.org/10.1007/BF01941137

[14] Nicolas Oury. 2005. Extensionality in the Calculus of Constructions. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Vol. 3603. Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293. https://doi.org/10.1007/11541868_18 Series Title: Lecture Notes in Computer Science.

[15] Jorge Luis Sacchini. 2011. *On type-based termination and dependent pattern matching in the calculus of inductive constructions.* Theses. École Nationale Supérieure des Mines de Paris. https://pastel.archives-ouvertes.fr/pastel-00622429

[16] Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *2013 28TH Annual IEEE/ACM Symposium on Logic in Computer Science (LICS) (IEEE Symposium on Logic in Computer Science).* IEEE, 345 E 47th St., New York, NY 10017 USA, 233–242. https://doi.org/10.1109/LICS.2013.29